THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le *jj/mm/2021* **par :**

Daniel LOCHE

Prévention des fautes temporelles sur architectures multicœur pour les systèmes à criticité mixte

---

JURY

Claire PAGETI	Ing. de Recherche, ONERA	Président du Jury
Sébastien FAUCOU	Maître de Conférences, Univ. de Nantes	Examinateur
Emmanuel GROLLEAU	Professeur, ISAE-ENSMA	Rapporteur
Liliana CUCU	Chargée de Recherche, INRIA	Rapporteure
Jean-Charles FABRE	Professeur, Toulouse INP	Directeur de Thèse
Michael LAUER	Enseignant-Chercheur, Univ. Toulouse 3	Co-directeur de Thèse
François GŒUSSE	Ingénieur, Renault SWLabs	Invité

École doctorale et spécialité :
EDSYS : Systèmes embarqués 4200046

Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes

Directeur(s) de Thèse :
Jean-Charles FABRE et *Michael LAUER*

Rapporteurs :
Liliana CUCU-GROSJEAN et *Emmanuel GROLLEAU*

Une Révolution est une évolution qui n'en a pas l'air.

——————— **Quelqu'un, quelque part.**

Remerciements

A faire en dernier :-)

Table des matières

Introduction

La complexité des systèmes cyberphysiques s'est accrue dramatiquement ces dernières décennies.

C'est ainsi que le domaine de l'automobile est successivement passé du tout mécanique a des architectures Électrique et Électronique (AEE) de plus en plus sophistiquées. Bien évidemment, cette tendance lourde s'appuyant sur les progrès des techniques numériques a permis de rendre aux clients des services plus avancés et pertinents qui ont gagné en intelligence. Cela s'est fait en s'appuyant tout particulièrement sur des aspects logiciels prépondérants en délaissant les anciens systèmes mécaniques ou électro-mécaniques.

Ces évolutions progressives dans la voiture ont mené à des ajouts de calculateurs ayant chacun son lot de fonctionnalités avancées, potentiellement accompagnées des capteurs (température de l'habitacle, présence sur les sièges...) mais aussi des actionneurs (système d'air conditionné, vitres, verrouillage centralisé...) nécessaires. C'est de cette façon que l'architecture distribuée dans l'automobile s'est étoffée pour atteindre jusqu'à 70 calculateurs dans un même véhicule. À terme, cette approche ne semble plus soutenable au vu de la demande en fonctionnalités supplémentaires liées aux technologies émergentes : le véhicule autonome et connecté.

C'est pour cette raison que la tendance d'ajout de calculateurs à une architecture distribuée toujours plus complexe est en train de s'inverser. C'est substitué par l'émergence de calculateurs multicœurs puissants qui peuvent se substituer a nombre d'ECU élémentaires. L'architecture actuelle s'oriente donc vers des architectures fédérées mettant en jeu des processeurs sur lesquels la coexistence d'applications critiques et non-critiques (niveau d'ASIL). Ces systèmes à criticité multiple induisent des problèmes de partage de ressources et de sûreté de fonctionnement.

Systèmes embarqués automobiles

Évolutions des systèmes embarqués

Système mécanique => système cyberphysique

Architectures EE

=> Augmentation complexité architecture EE => augmentation des besoins (puissance de calcul, ADAS, voiture connectée/autonome...)

Tendances et Contraintes actuelles

Tendances

=> Nouvelles architectures EE fédérées, virtualisation + multi-coeurs Présentation des différents risques d'interférence multicoeur => Évolutivité (Adaptive AUTOSAR, car as a service)

Contraintes et limitations

Difficulté de transition Complexité Coûts

Objectif(s), contribution et Problématique

Transition partie I - enjeux des fautes temporelles à cause des tendances

Chapitre 1

Contexte et Enjeux des Systèmes Embarqués

Sommaire

La conception des systèmes embarqués, typiquement automobiles, a subi de fortes évolutions orientées vers de nouvelles fonctionnalités centrées sur le logiciel. Ces évolutions demandent des capacités de calcul de plus en plus importantes et donc des architectures matérielles pour supporter la demande grandissante en fonctionnalités. Par ailleurs le contexte industriel mène à la disparition des calculateurs d'antan, monocœurs, pour se focaliser sur des calculateurs plus complexes et puissants, multicœurs. Cette tendance au multicœur provient à la fois d'une limitation technologique et d'un besoin grandissant : la façon d'augmenter les capacités de calculs par les méthodes classiques (montée en fréquence) atteint ses limites et les capacités d'exécution concourante de logiciel est de plus en plus demandée dans un contexte aux contraintes financières et de *time-to-market* fortes. C'est ainsi que né la volonté de passer sur des architectures électriques et électroniques plus centralisées via l'utilisation d'une quantité réduite d'unités de calcul, mais intégrant un plus grand nombre de fonctionnalités de traitement en leur sein ; en un mot, des processeurs multicœurs. Cette volonté implique cependant une superposition des difficultés inhérentes aux architectures matérielles plus complexes avec les contraintes de sûreté de fonctionnement du logiciel. Nous faisons donc face à des systèmes à criticité mixte exécutés sur des calculateurs aux mécanismes complexes. Nous verrons ainsi dans ce

chapitre quels sont les aspects essentiels de ce contexte et ses spécificités à prendre en compte pour proposer de nouveaux éléments de réponse dans la conception de systèmes à criticité mixte sur processeurs multicœurs. Nous conclurons cette partie avec la présentation de la problématique à laquelle nous tenterons de répondre ainsi que la présentation des différents chapitres de cette thèse.

1.1 Évolutions des Systèmes embarqués

1.1.1 Nouveaux systèmes intelligents et connectés

Si l'on prend le cas du domaine automobile, depuis près de trente ans l'industrie n'a cessé de faire évoluer la façon de concevoir les véhicules et notamment leurs systèmes sous-jacents. Comme illustré avec le diagramme 1.1, la transition s'est faite de modifications purement mécaniques vers des évolutions électriques, puis électroniques et de plus en plus intelligentes. Les systèmes de divertissement du consommateur ont été les premiers, dans le milieu des années 1920, à introduire des composants électroniques au sein des véhicules sous la forme de récepteurs radio à lampes ! Si l'apparition de transistors, dans les années 1950, a contribué à l'amélioration des capacités techniques des appareils et à la diffusion massive des autoradios au sein des automobiles, le concept de base a peu évolué jusqu'à la fin des années 1970. L'introduction des premiers systèmes de navigation dans les années 1980, puis des systèmes multimédia dans les années 2000 a changé la donne. Désormais, l'ancienne façade de l'autoradio devient un écran de commande nommée *head unit* et concentre 70% du code du véhicule. Les voitures se sont modernisées avec l'ajout de calculateurs dédiés à des fonctions internes ou des services. Le développement des technologies de l'industrie 4.0 mène à une augmentation exponentielle du logiciel embarqué dans l'automobile au cours des 15 dernières années [Blanchet 2016], avec la présence de plus de 60 calculateurs embarqués dans certains modèles. Les contrôles mécaniques et autres systèmes électriques "simples" cèdent la place au monde du numérique. Les équipements électroniques et logiciels se multiplient au sein du véhicule pour l'aide à la conduite (*Advanced Driver-Assistance System – ADAS*) et l'ajout de services [Schmidt 2010]. De fait, le système multimédia moderne a un rôle qui va bien au-delà de celui du simple autoradio : il devient l'interaction principale entre le consommateur et le véhicule et devient un critère de choix prépondérant à l'achat.

Ainsi, du simple Système Anti-blocage des roues (ABS), on a introduit des Assistants à la Conduite tels que le Freinage d'Urgence (*Emergency Braking System*) ou encore le Système de Gestion de Ligne (*Lane Support System*) qui permet à la fois l'Avertissement de Dépassement de Ligne (*Lane Departure Warning*), l'Assistant de Maintien de Ligne (*Lane Keeping Assist*) et le Maintien de Ligne d'Urgence (*Emergency Lane Keeping*)... et il ne s'agit là que de 2 fonctionnalités supplémentaires ! En parallèle, la voiture devient de plus en plus automatisée, voire autonome. Elle gagne en connectivité avec la prise en compte de données extérieures possiblement avec un lien direct au cloud pour proposer une diversité de services : météo, divertissement, trafic routier, pour n'en citer que quelques exemples. Les systèmes embarqués deviennent par conséquent aussi connectés. On parle de communications

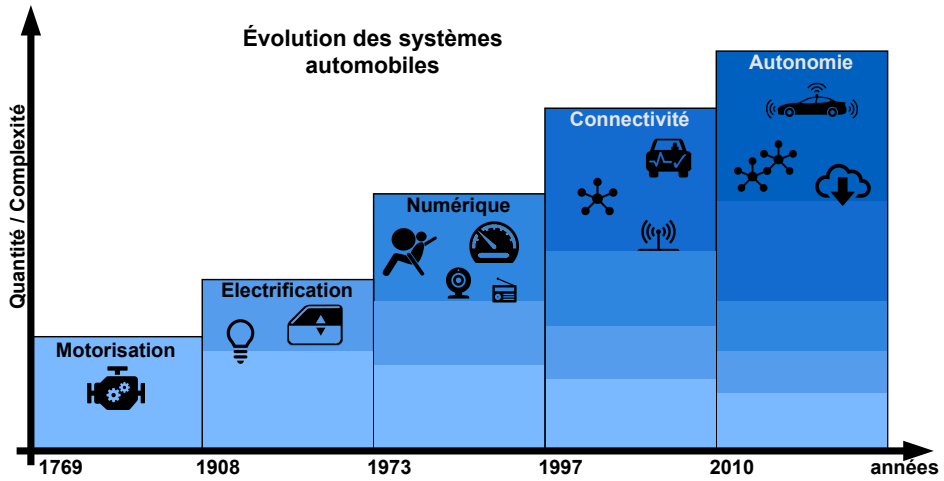


FIGURE 1.1 – Principaux domaines d'évolution des systèmes automobiles au fil du temps

car-to-car entre véhicules ou *car-to-infrastructure* entre véhicule et infrastructures routières par exemple. Cette ouverture du système à son environnement est à double tranchant. D'une part cela offre de nouveaux horizons de fonctionnalités et optimisations de conduite, avec des possibilités d'évolutivité simplifiée. Mais d'autre part la complexité va grandissante avec les enjeux d'ingénierie que cela implique.

De façon plus générale, le contexte industriel actuel fait émerger de nouvelles technologies basées sur des logiciels de plus en plus complexes et performants. Cela est rendu possible via l'émergence d'architectures matérielles toujours plus puissantes et performantes. Ces améliorations permettent le développement et la mise en application de nouvelles technologies comme les réseaux de communication sans fil haute performance ou encore l'usage d'intelligences artificielles. On retrouve ainsi un nombre grandissant de fonctionnalités directement embarquées dans l'automobile, l'avion, le train pour répondre à la fois à de nouveaux besoins fonctionnels : assistance à la conduite/pilotage, tableaux de bord, etc. et à des besoins de confort d'usage : info-divertissement, connectivité, automatisations...

D'un point de vue logiciel, les mises à jour de systèmes embarqués incluent à la fois de nouvelles fonctionnalités critiques pour le bon fonctionnement du système, mais aussi l'ajouts de fonctions moins critiques. Ces mises à jour de services non essentiels amplifient la multiplicité des niveaux de criticités du logiciel embarqué et donc la cohabitation entre sous-systèmes critiques et sous-systèmes non-critiques que l'on pourrait qualifier de "confort".

D'un point de vue matériel, il y a de fortes convergences sur les architectures employées dans les différents domaines. Historiquement, on retrouvait en premier lieu des calculateurs monocœurs. Cependant, les diverses évolutions d'exigences ont fait apparaître des limites en capacité de calcul. La montée en fréquence de fonctionnement atteint un seuil maximum à cause de la chauffe et la consommation que cela implique. Tandis que l'augmentation du nombre de transistors qui composent les processeurs arrive aux abords des limites physiques : la taille de gravure du silicium

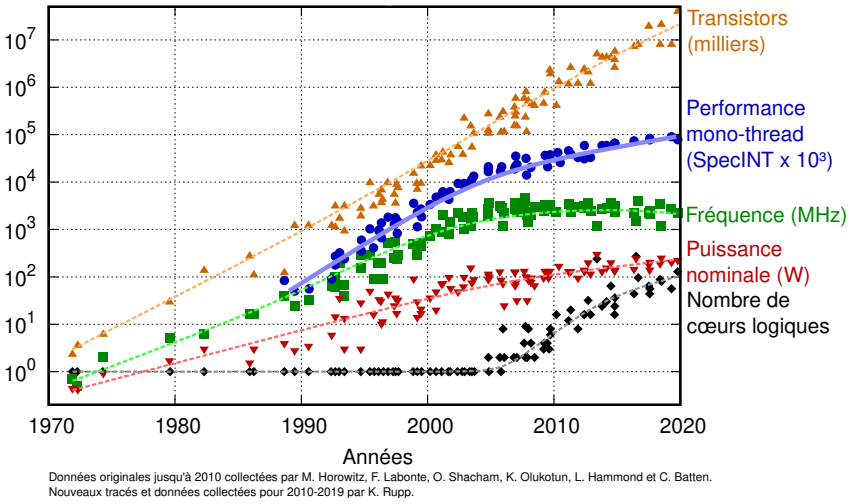


FIGURE 1.2 – 42 Ans d'évolution des processeurs - Tendances

arrive au même ordre de grandeur que la taille des atomes de silicium dont elle est composé. De fait, jusqu'à récemment encore, la Loi de Moore [Thompson 2006] sur la puissance des processeurs s'est vérifiée. Des premiers microprocesseurs Intel en 1971, avec quelques milliers de transistors de 10 µm, l'on est aujourd'hui à plus de 1 milliards de transistors de près de 10 nm. Mais à l'aune d'une gravure proche des 2 nm, on environne les dimensions de 10 à 15 atomes et les effets de la physique quantique entrent en jeux. Par conséquent, l'on se dirige vers les limites des technologies actuelles pour poursuivre ces améliorations de puissance. Pour ces raisons, le plus grand levier de progression disponible aujourd'hui repose sur la parallélisation des unités de calcul, et donc la notion de calculateur multicœur, qui est apparue dès les années 1950 [Smotherman 2005]. Les fondeurs s'orientent vers des processeurs où la montée en puissance est assurée par la multiplication des unités de calcul (dit "cœurs") parallèles dans le processeur. On passe ainsi de monocœurs toujours plus petits et compacts à des duals/quadri cœurs... et l'on va aujourd'hui jusqu'à des supercalculateurs à plus de 128 cœurs. Tous ces changements se visualisent parfaitement avec l'évolution des caractéristiques des processeurs au fil des années en tel qu'agrégé par K. Rupp [Rupp 2020]. Cette évolution est la bienvenue dans tous les secteurs concernés, allant du grand public dans les ordinateurs, téléphones et autres multimédias jusqu'aux applications industrielles en passant par les usages de serveurs réseaux et centres de calculs.

Il existe divers types d'architectures matérielles parmi les évolutions multicœurs que l'on retrouve aujourd'hui. On pourrait de façon simple différencier entre les multicœurs classiques, les manycœurs et à l'extrême ce que l'on connaît sous le nom de GPU, les processeurs graphiques.

Multicœurs "classiques" Les calculateurs multicœurs "classiques" disposent d'un certain nombre d'unités de calculs ("cœurs"), auxquelles sont adjointes diverses zones mémoires (cache, RAM, ROM). Le tout est piloté par des contrôleurs

et bus de transferts de données pour interconnecter les cœurs, les cellules mémoires et les entrées/sorties. Dans les versions les plus récentes, des modules dédiés peuvent être ajoutés pour des fonctionnalités spécifiques comme le chiffrement.

La mémoire est partagée à différents degrés entre les cœurs. De façon à décongestionner les accès mémoire et accélérer ces dernières, une hiérarchie mémoire est mise en place, associant des espaces mémoire progressivement plus petits et rapides en fonction de leur proximité au processeur. Il s'agit ici de trouver un équilibre entre coût de la mémoire et vitesse d'accès aux données. En effet, cette dernière dispose de trois caractéristiques antagonistes :

— la **latence** - temps d'accès aux données,

— la **bande passante** - débit de données accessible,

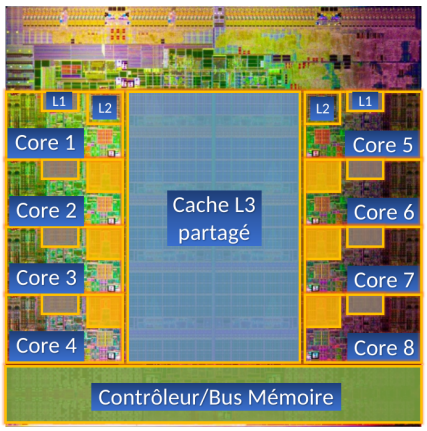
— la **taille** mémoire - espace mémoire disponible (pour un coût donné).

Un espace mémoire pourra être soit de petite taille, mais rapide au niveau de son temps d'accès, soit de grande taille et plus lent comme schématisé dans la Figure 1.3b. On a par conséquent au plus proche des cœurs les registres, de taille très limitée (octets) mais au temps d'accès très rapide : ils sont la base pour toutes les opérations effectuées par le processeur. À l'opposé, la mémoire principale, de très grande taille (Go/To) pour laquelle tous les cœurs doivent passer par un bus commun pour y accéder. C'est donc la mémoire la plus lente d'accès mais aussi la moins coûteuse. Plusieurs intermédiaires ont été mis en place entre ces deux types de mémoire. Il s'agit typiquement de niveaux de cache qui peuvent être non partagés, c'est-à-dire propres à chaque cœur ou bien commun à tous. Le dernier niveau de cache, partagé, est classiquement appelé LLC (*Last Level Cache*) et donne la limite entre les espaces mémoire limités en cache avec des accès rapides d'une part et la mémoire principale qui va provoquer de grands ralentissements d'autre part. On retrouve ainsi avec l'exemple de la Figure 1.3a un cas de calculateur multicœur basé sur le cache, avec 8 cœurs, des niveaux de cache mémoire séparés (L1 et L2) et partagé (L3) ainsi que le bus d'accès à la mémoire principale.

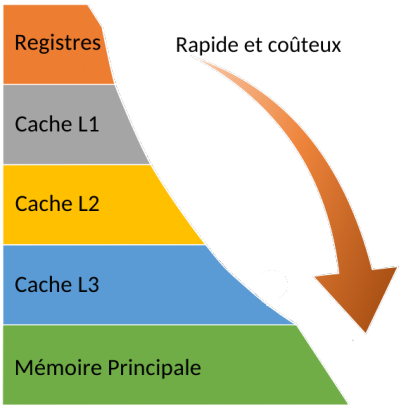
La gestion du contenu de ces caches (en lecture et écriture) est géré par une politique d'accès mémoire. Cette politique est essentielle à un usage efficace des caches du fait de leur espace limité qui demande à faire des choix sur son usage. Cela est peu documenté par les constructeurs, et chacun aura sa façon de faire. La méthode de base la plus répandue étant empirique, par principe de localité temporelle [Durrieu 2014] et spatiale [Wilkes 1965]. On considère que plus une donnée a été récemment accédée, plus elle a de chance d'être à nouveau utilisée. De même si une donnée est sollicitée, alors les données proches spatialement ont aussi plus de chance d'être utilisées. Nous n'iront pas plus dans les détails sur les politiques de gestion d'accès à la mémoire. Il faut garder à l'esprit qu'elle est plutôt subie par les industriels qui intègrent le matériel dans leurs systèmes. Pour un processeur donné on aura certaines performances de calcul et accès mémoire, et il faudra mettre en comparaison les performances "par défaut" d'un logiciel sur une architecture donnée face au même système, pas clair ?

mais avec des surcouches de gestion du logiciel apportées par l'intégrateur

.



(a) Exemple d'architecture multicœur

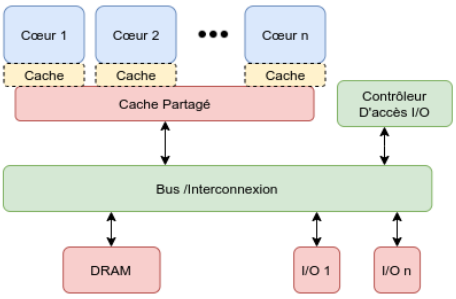


(b) Schématisation de la hiérarchie mémoire selon leur coût et performance

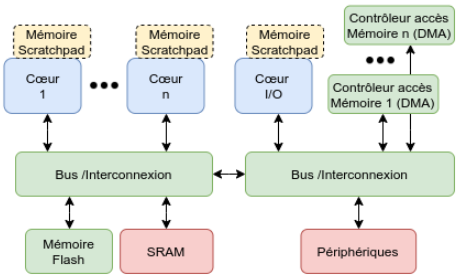
FIGURE 1.3 – Exemple multicœur et mémoires

Il existe des variations d'architectures différentes selon les fondeurs, que l'on peut classifier en deux catégories principales. D'une part les multicœurs basés sur le cache (comme celui susmentionné) et d'autre part les multicœurs basés sur *scratchpad*, c'est-à-dire des mémoires locales dédiées à chaque cœur. On peut voir la différence fondamentale de structure entre ces deux variations sur la Figure 1.4. **TBA**

Voir où et comment parler des multicœurs scratchpad ! Potentiellement dans "Solutions existantes".



(a) Architecture calculateur multicœur basé sur le cache



(b) Architecture calculateur multicœur basé sur Scratchpad

FIGURE 1.4 – Exemple multicœur et mémoires

Calculateurs manycœurs Les calculateurs manycœurs sont des microprocesseurs incluant un grand nombre de cœurs dans l'objectif primaire d'une plus grande capacité d'exécution de code parallèle. Pour ce faire, les cœurs peuvent être spécialisés avec la réduction des instructions réalisables et optimisations à des tâches spécifiques. C'est la différence principale avec les multicœurs qui possèdent en en général des cœurs identiques (processeur homogène) avec de bonnes performances

à la fois en série et en parallèle. Les architectures manycœurs grâce à leurs spécificités demandent des méthodes de programmation appropriées pour pouvoir être pleinement exploités dans le cadre d'une application. Cela augmente donc le niveau de complexité de développement, mais au bénéfice d'une forte amélioration des performances.

Les GPUs (*Graphic Processing Unit*) sont un cas particulier de manycœurs à présent très répandu pour des usages variés [Owens 2008]. Cette forte expansion des GPU est due non seulement aux capacités de rendu graphique, mais surtout à leurs capacités de programmation parallèle poussée au maximum. Un grand nombre de domaines, notamment dans la recherche, y voient donc un microprocesseur d'usage général à hautes capacités de calcul parallèle. Les GPU sont efficaces du fait qu'ils permettent de réaliser le même calcul sur un très grand nombre de données différentes (typiquement calculs matriciels) pour obtenir tout autant de résultats en sortie. Il s'agit d'un modèle dit *SIMD - Single-Instruction, Multiple-Data.* Là où les multicœurs conventionnels se focalisent sur des cœurs versatiles qui s'adaptent pour pouvoir gérer tous les cas d'applications, les GPU se focalisent sur la réalisation de tâches identiques en parallèles, ils restent donc spécialisés pour des types de tâches spécifiques, en complément de processeurs plus polyvalents.

Dans le cadre de ces recherches, nous nous focaliseront sur le dénominateur commun le plus utilisé dans les architectures électriques et électroniques, qui est donc le processeur multicœur basé sur le cache.

1.2 Risques et Problématique

Dans le cadre du contexte automobile, on se dirige vers un nouveau paradigme, où la voiture n'est plus un système mécanique sur lequel on adjoint du logiciel, mais à l'inverse un superordinateur multifonctionnel auquel on implante des roues et un moteur. Les systèmes automobiles sont ainsi devenus des systèmes cyberphysiques qui entrent en interaction à la fois avec les utilisateurs et l'environnement. On distingue deux grands domaines de logiciels embarqués dans le véhicule. Tout d'abord l'info-divertissement, qui réunit les systèmes multimédias et autres affichages non nécessaires à l'usage primaire du véhicule. Et deuxièmement les calculateurs enfouis qui réalisent des fonctions essentielles qui ne sont pas nécessairement visibles de l'utilisateur, telles que le contrôle moteur. Pour soutenir ces besoins émergents, il est nécessaire de se baser sur des architectures matérielles plus puissantes comme les multicœurs. Cependant, cette disruption apporte de nouveaux enjeux, notamment de sécurité, vie privée, mais aussi sur la prédictibilité et la sûreté de fonctionnement du système à cause de sa complexification. Cela fait donc évoluer les systèmes embarqués dans un environnement profondément à risques, mais qui en plus s'accompagne de contraintes fortes. Nous nous devons donc d'introduire ici les notions de Sûreté de fonctionnement nécessaire à l'analyse.

1.2.1 Sûreté de Fonctionnement Informatique

La sûreté de fonctionnement d'un système informatique (SdF) est "*la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre, le service étant le comportement du système perçu par un utilisateur, cet utilisateur étant un système (informatique, humain, environnemental) qui interagit avec le premier.*" [Laprie 1996]. C'est donc la capacité d'un système informatique de répondre de manière correcte, conformément aux spécifications fonctionnelles, à une requête d'un autre système. La sûreté de fonctionnement est définie en fonction de trois notions principales : **a)** les *attributs* qui définissent les propriétés assurées, **b)** les *entraves* qui caractérisent les circonstances indésirables mais prévues, **c)** et les *moyens* qui précisent les techniques permettant au système de fournir son service . Selon les services souhaités par l'utilisateur, ce dernier peut vouloir accentuer certaines propriétés pour assurer le bon fonctionnement du système. Ainsi la sûreté de fonctionnement englobe les attributs suivants :

— La **disponibilité** - la capacité d'être prêt à délivrer le service correct ;

— La **fiabilité** - l'assurance de continuité d'un service correct ;

— La **sécurité-innocuité** - l'assurance de non-propagation de conséquences catastrophiques à l'utilisateur ou l'environnement ;

— L'**intégrité** - l'assurance de non-altération du système ;

— La **maintenabilité** - l'aptitude à la réparation et à l'évolution du système.

Ces attributs permettent d'une part d'exprimer les propriétés devant être respectées par le système, et d'autre part d'évaluer la qualité du service délivré vis-à-vis de ces propriétés. Les aspects de sécurité, au sens de la confidentialité et des attaques face à des actions malveillantes indésirables ainsi que la confidentialité, c'est-à-dire, la non-divulgation d'information non autorisée, ne seront pas abordés dans cette thèse.

Les entraves à la sûreté de fonctionnement sont les défaillances, les erreurs et les fautes. Une défaillance est une transition d'un service correct vers un service incorrect. Un service est considéré incorrect s'il n'est pas conforme à la spécification ou si la spécification ne décrit pas avec précision la fonction du système. Étant donné qu'un service consiste en une séquence d'états externes du système (observés par l'utilisateur), la survenue d'une défaillance signifie qu'au moins un des états externes s'écarte de l'état correct du service. La déviation est liée à une erreur, qui représente la partie de l'état interne du système pouvant entraîner une défaillance, dans le cas où elle atteint l'interface du service du système. La cause déterminée ou présumée d'une erreur est appelée une faute. La Figure 1.5 représente ce lien de cause à effet. Le fait de prévenir la causalité entre fautes est défaillances pour le bon fonctionnement se désigne par la méthode de silence sur défaillance. C'est-à-dire qu'une faute ou une erreur n'aura pas plus de conséquences et ne provoquera pas outre de défaillance, ou inversement.

Pour minimiser l'impact de ces entraves, la sûreté de fonctionnement dispose de méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement des fonctions du système. Le développement d'un système sûr de fonctionnement passe donc par l'utilisation combinée de ces méthodes, appelés moyens, pouvant être classées en quatre types :

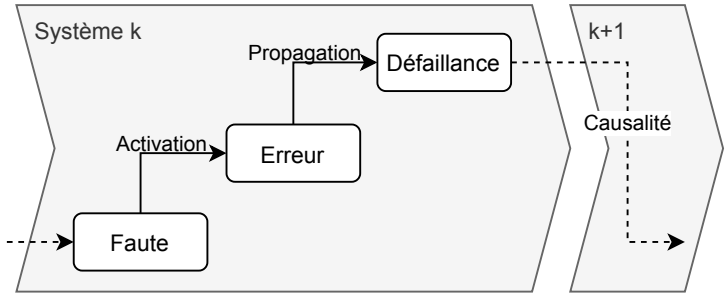


FIGURE 1.5 – Sûreté de fonctionnement - chaîne de causalité

— **Prévision** des fautes : estimation de la présence, de la création et des conséquences des fautes (p. ex. Analyse FMEA) ;
— **Prévention** des fautes : méthodes visant à réduire les occurrences ou l'introduction de fautes (p. ex. outil de génie logiciel, processus de développement strict) ;
— **Élimination** des fautes : réduction du nombre et de la sévérité des fautes (p. ex. test, injection de fautes) ;
— **Tolérance** aux fautes : capacité de fournir un service, optimal ou dégradé, en présence de fautes (p. ex. techniques de redondance).

La prévention et la tolérance aux fautes visent à fournir la capacité de délivrer un service correct, tandis que l'élimination et la prévision des fautes visent à susciter la confiance en cette capacité en justifiant que les spécifications fonctionnelles de sûreté de fonctionnement et de sécurité sont adéquates et que le système est conforme. Toutes ces techniques sont dédiées à garantir des propriétés de sûreté de fonctionnement issues de spécification non fonctionnelles.

Tolérance aux Fautes Les fautes auxquelles un système doit faire face sont nombreuses et peuvent ne pas avoir d'impact sur celui-ci tant qu'un ou plusieurs évènements ne se sont pas produits. On les appelle alors des fautes dormantes. Une fois activées, ces fautes peuvent avoir un impact catastrophique sur le système. D'origines diverses et variées, certaines fautes sont dues à l'environnement, au matériel, ou encore à l'être humain.

Chaque faute peut provoquer une ou des erreurs différentes pouvant entraîner la défaillance du système. Malgré l'application des techniques de prévention et d'élimination des fautes, certaines subsistent et sont à même d'être activées.

Un système tolérant aux fautes doit pouvoir assurer à l'utilisateur un service correct en dépit des fautes pouvant altérer ses composants, durant sa conception ou son interaction avec d'autres systèmes [Avizienis 2004]. La Tolérance aux fautes est mise en œuvre grâce aux moyens de **détection** d'erreurs, c.-à-d., l'identification des déviations du service correct, et de **recouvrement**, c.-à-d., les techniques permettant en cas d'erreur détectée de passer d'un état de système fautif à un état assurant un service nominal ou dégradé.

La détection d'erreur peut être soit concurrente et se déroulant pendant l'exécution du système soit anticipée en vérifiant les paramètres du système lors de

la suspension de son exécution. Une fois cette erreur détectée, les techniques de recouvrement peuvent être employées, d'une part pour assurer le service désiré et éviter la propagation de l'erreur (traitement des erreurs) et d'autre part pour isoler le composant fautif, diagnostiquer l'erreur, trouver et déterminer la faute originelle pour assurer une opération de maintenance (traitement des fautes).

Les techniques de détection et de recouvrement sont nombreuses et sont regroupés dans des mécanismes de tolérance aux fautes associés à un ou plusieurs types de fautes. Il n'y a à l'heure actuelle aucun mécanisme générique pouvant pallier n'importe quel type de fautes ou d'erreurs. Que cela soit de la redondance matérielle, logicielle, temporelle, de la diversité dans l'implémentation ou l'architecture, les techniques sont nombreuses et souvent propres à chaque domaine et au budget alloué à la tolérance aux fautes.

Dans le cadre de ces travaux de recherche, nous nous intéresseront particulièrement à la tolérance aux fautes qui attrait donc à la bonne exécution de tâches hébergées au sein d'un même calculateur. Dans le contexte industriel susmentionné, un même calculateur exécute des tâches pour des fonctionnalités variées et par conséquent avec des niveaux de criticité variés. Cela engendre notamment des contraintes sur les temps d'exécution des logiciels les plus critiques. C'est ce qu'on qualifie de systèmes temps réel. Nous sommes en résumé dans un contexte à criticité mixte, où du logiciel de système temps-réel va coexister avec du logiciel avec des contraintes temporelles moins strictes, voire aucune contrainte. L'implémentation de mécanismes de sûreté de fonctionnement dans ce contexte-là relève alors de la gestion de fautes temporelles dans un système à criticité mixte.

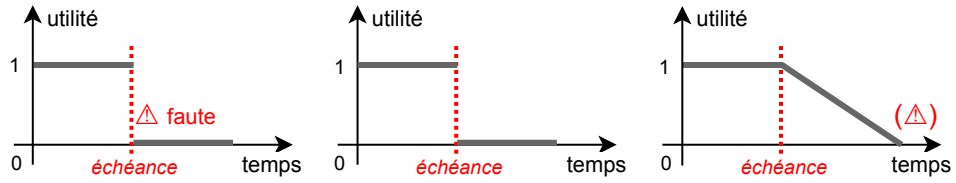
1.2.2 Systèmes temps-réel et Ressources partagées

Système temps-réel Les systèmes embarqués sont conçus sur la base d'un modèle de capteurs et actionneurs. Les capteurs représentent l'ensemble des éléments qui permettent d'obtenir les données d'entrée au système de façon à ce qu'il puisse réaliser sa fonction. Il s'agit notamment des informations de l'environnement du véhicule, mais aussi des données internes avec tout l'état de fonctionnement actuel ainsi que les interactions avec l'utilisateur. Ces informations sont alors gérées par les calculateurs de décision via des algorithmes plus ou moins complexes. Le logiciel permet donc à partir de ces données d'entrée de calculer les commandes qui sont dirigées vers les actionneurs. Les actionneurs sont alors en bout de chaîne afin d'accomplir la commande. Dans le cas où les données d'entrée fournies par les capteurs sont liées aux données de sortie, on parle alors d'une *boucle* de contrôle. Typiquement avec le chauffage d'un logement qui utilise un capteur de température pour une consigne de température donnée.

Prenons un exemple hypothétique de contrôle de l'injection moteur pour une voiture. En entrée, le calculateur de contrôle moteur récupère entre autre les informations du capteur de vitesse de rotation du moteur, de la quantité d'essence en réservoir et l'accélération demandée par le conducteur. Il peut alors calculer l'instant et la quantité de carburant qu'il sera nécessaire d'injecter dans le moteur. Cette commande est alors transmise à l'actionneur, l'injecteur, pour être réalisée. Et ce bloc de contrôle-commande doit se répéter périodiquement pour suivre la consigne

tout le long de l'utilisation du véhicule.

Tous ces éléments de contrôle-commande ont en commun d'avoir des contraintes temporelles. Le temps de réaction –qui définit la durée entre la récupération des données des capteurs jusqu'à la réalisation de la commande par les actionneurs– peut alors être une donnée critique pour certaines applications comme l'exemple donné ci-dessus (*inutile de dire qu'un contrôle d'injection moteur qui prend trop de temps à déterminer combien de carburant injecter aura des conséquences bien évidemment indésirables...*). Ainsi, ce genre d'applications nécessitent à la fois de retourner des résultats corrects mais aussi de les délivrer dans les temps.



(a) Temps-réel dur (b) Temps-réel ferme (c) Temps-réel mou

FIGURE 1.6 – Modèles d'utilité des résultats d'une tâche temps-réel

Plus généralement, les applications embarquées se caractérisent par un ensemble de tâches logicielles qui interagissent entre-elles. Elles sont soit périodiques (c.-à-d. exécutés à intervalle réguliers) soit apériodique (sur réception d'un événement). Chaque tâche possède ses spécifications propres en termes de données d'entrée, de sortie ainsi que ses paramètres d'exécution (selon les cas : période, niveau de priorité, allocation physique) dont une échéance d'exécution. Les systèmes temps-réel peuvent se catégoriser en 3 catégories qui sont schématisées en Figure 1.6. On retrouve d'une part les systèmes **temps-réel strict** ("*hard real-time*") où le respect de l'échéance est strict en Figure 1.6a. Il est alors considéré qu'une tâche dont le temps de réponse dépasserait l'échéance serait une faute temporelle indésirable et les données renvoyées par la tâche n'ont plus de valeur. Le même modèle mais sans conséquences après dépassement de l'échéance est nommé **temps-réel ferme** ("*firm real-time*") illustré en Figure 1.6b. À l'inverse, les systèmes **non-temps-réel** n'imposent pas de contraintes d'échéance sur l'exécution des tâches. Il s'agit donc de faire au mieux, mais tout dépassement des temps d'exécution nominaux n'a pas de répercussions. C'est ce que l'on côtoie couramment via nos appareils de tous les jours comme le smartphone ou l'ordinateur. Enfin, les systèmes **temps-réel souple** ("*soft real-time*") sont un entre-deux où l'échéance représente un seuil limite au-delà duquel la valeur de retour de la tâche garde une utilité pour le système mais qui décroît avec le temps, jusqu'à ne plus être pertinente comme représenté en Figure 1.6c. On dit alors que la donnée est "périmée". Ce dépassement peut alors provoquer ou non une faute.

Les analyses d'exécution temporelles des tâches constituent alors un aspect essentiel du développement de logiciel critique afin de garantir le respect des échéances. Cela peut se faire soit de façon expérimentale ou théorique. L'objectif étant de vérifier l'ordonnancement, c'est-à-dire la bonne gestion de l'exécution du logiciel sur le processeur suivant les contraintes imposées (échéances, dates d'activation,

périodes...). Un système est dit prédictible si l'on est capable de prouver de façon théorique que les contraintes temporelles seront respectées. Cela se fait par le biais d'analyses d'ordonnançabilité. Un technique classique de ce type d'analyse consiste à évaluer les pires temps d'exécution ("*Worst-Case Execution Time – WCET*"). Le WCET indique la durée maximum au-delà de laquelle on sait qu'en toutes conditions, la tâche correspondante aura terminé son exécution. Il est possible de comparer les WCET des tâches avec leurs échéances. Pour du temps réel strict, les valeurs de WCET se devront d'être strictement inférieures aux échéances, là où pour du temps réel ferme ou souple on pourra se contenter d'estimation ou de résultats statistiques.

Au sein d'un même processeur, toutes les tâches n'auront potentiellement pas les mêmes types de contraintes d'exécution. Mais en plus de cela, les architectures matérielles multicœurs complexifient d'autant plus l'analyse.

Ressources Partagées Comme nous l'avons vu précédemment sur l'architecture multicœur, il existe un bon nombre de ressources qui sont partagées entre les différents logiciels qui sont exécutés. Ces partages de ressources peuvent influencer directement sur l'exécution des tâches et donc sur leur capacité à respecter les échéances. En effet, si plusieurs tâches ont des besoins concurrents d'accès à une même ressource alors nécessairement l'une va passer avant l'autre. Cette dernière sera *de facto* retardée dans son exécution. Il existe ainsi de nombreuses sources de retards potentiels d'exécution [Kotaba 2013] :

— Mémoire –
  - erreurs en lecture : si une donnée n'est plus présente en cache du fait qu'elle a été remplacée par les données d'une autre application. Cela demande alors à remonter sur les niveaux de mémoire supérieurs, ce qui engendre des temps d'accès supplémentaires importants ;
  - accès concurrents : l'accès concurrent à un niveau de mémoire partagée se fait par le bien d'un contrôleur d'accès mémoire, qui va devoir arbitrer sur l'ordre et le temps alloué à chaque tâche.
  - Cohérence mémoire : selon les technologies de gestion de cache utilisées, il faut gérer la cohérence mémoire. Si une donnée est utilisée dans plusieurs mémoires non partagées, alors il faut s'assurer que la donnée en question reste cohérente entre toutes les tâches qui s'en servent. Cela implique en général une synchronisation sur les niveaux de mémoire supérieure quand elle est modifiée de façon locale, et inversement une propagation des modifications vers les tâches qui manipulent la donnée.

— Périphérique et I/O en général – chaque périphérique dispose de son propre contrôleur d'accès. On a donc les mêmes enjeux qu'avec les accès concurrents à la mémoire dans le cas où plusieurs tâchent utilisent la même entrée/sortie. Le cas principal ici pour une architecture embarquée est sur l'utilisation d'un bus de communication externe qui sert à interconnecter les calculateurs. L'envoi et la lecture de message sur de tels bus de communication peuvent alors engendrer un grand nombre d'usages concurrents.

— Bus d'interconnexion – Le fonctionnement même des processeurs implique l'utilisation de bus internes afin de gérer la transmission et le stockage de

données. Tout usage de ces bus d'interconnexion peuvent alors impliquer des usages concurrents qui impactent l'accès aux données des tâches.

— Puissance de calcul – Enfin, mais pas des moindres, il n'y aura probablement jamais autant de cœurs que de tâches sur un processeur multicœur. Il est de ce fait évident que les tâches devront se partager tout ou partie des cœurs selon leur allocation physique. C'est là que va entrer en jeu la stratégie d'ordonnancement des tâches. La politique d'ordonnancement joue un rôle essentiel pour permettre le respect des contraintes temporelles et optimiser l'usage de la puissance de calcul pour limiter au maximum les temps d'attente en file d'exécution des tâches.

1.2.3 Problématique et Objectifs

Dans le contexte industriel qui concerne notre étude, les évolutions des systèmes cyberphysiques présentés précédemment impliquent que des tâches de différents modèles d'exécution doivent être intégrées au sein d'un même multicœur. On parle alors de système à criticité mixte. Cette coexistence de fonctionnalités va augmenter la complexité d'étude de sûreté de fonctionnement afin de garantir l'ordonnançabilité des tâches et le respect des contraintes temporelles. Et comme nous venons de le voir, les nouveaux calculateurs multicœurs ajoutent en niveau de complexité avec l'augmentation de risque d'interférence entre logiciels concurrents. Il devient par conséquent de plus en plus complexe de mener des études théoriques pour estimer les pires temps d'exécution et donc l'ordonnançabilité des tâches. La conséquence directe à cela est un manque de garanties claires sur le bon respect des échéances temporelles pour les tâches les plus critiques pour lesquelles on ne peut se permettre de telles fautes.

On verra qu'il existe de nombreuses méthodes qui permettent de réduire les interférences et donc fiabiliser les études d'ordonnançabilité. Cependant, cela se fait en général au prix d'un compromis sur les performances de calcul. Hors, c'est pour cette même puissance de calcul que la transition vers des calculateurs multicœurs s'est faite. Il semble alors essentiel de vouloir l'exploiter au maximum. On a deux objectifs qui s'opposent, mais qui sont tout autant essentiels. D'une part l'exploitation au maximum des capacités de calcul pour héberger tout le logiciel nécessaire aux nouvelles fonctionnalités des systèmes embarqués. D'autre part continuer à donner des garanties fortes de respect des contraintes temps réel pour les tâches critiques.

Cela nous mène donc à la problématique centrale de cette thèse, qui est d'identifier les leviers et mécanismes qui peuvent permettre d'atteindre au mieux les deux objectifs susmentionnés d'optimisation de l'usage du processeur avec les garanties temporelles liées aux systèmes critiques. Nous tenterons dans la suite de proposer une réponse à cette problématique par le biais d'une nouvelle approche qui mène à l'usage d'un mécanisme de surveillance et de contrôle de l'exécution des tâches pour éviter toute faute temporelle en cas d'occurrences d'interférences tout en permettant par ailleurs de libérer toute la puissance de calcul disponible dans l'exécution des tâches.

1.3 Contraintes et Hypothèses

1.3.1 Contexte industriel Automobile

Cette problématique s'inscrit dans un contexte industriel aux contraintes spécifiques. Il est donc important d'avoir ces éléments en ligne de compte pour proposer une analyse et des contributions pertinentes. Historiquement dans le domaine automobile, les calculateurs embarqués étaient conçus de manière *ad hoc*. Le logiciel et le matériel étaient intimement liés. Cela conduit à un nombre de calculateurs très important, chaque calculateur apportant une fonctionnalité qui lui est propre. Les architectures se composent alors d'un grand nombre d'unités de calcul interconnectées. Ce type d'architecture distribuée présente des inconvénients évidents en terme d'évolutivité du système et de coût de développement. À chaque changement de support physique le logiciel doit passer par un nouveau stade de développement plus ou moins conséquent. Inversement, une mise à jour du logiciel ou un ajout de fonctionnalité demande une prise en compte de l'intégration matérielle avec potentiellement des modifications matérielles pour suivre les évolutions. Chaque ajout de fonctionnalités va de cette façon ajouter de nouveaux calculateurs dédiés, complexifiant d'autant plus l'architecture.

Toutes ces contraintes de développement s'inscrivent dans un contexte bien cadré par des normes et standards. L'architecture fédérée telle qu'elle arrive dans les architectures électriques et électroniques abolit la séparation physique qui préexistait entre les composants logiciels, par leur agrégation dans un nombre réduit de calculateurs plus puissants. Cela résulte en un accroissement de la complexité de l'intégration et de la mise en œuvre de la sûreté de fonctionnement.

1.3.2 Standards industriels et Concept de Criticité

Les processus de développement de systèmes embarqués sont régis par des standards qui donnent des garanties sur le bon fonctionnement et donc vis-à-vis du respect des contraintes non fonctionnelles. Ces standards recommandent des directives de développement qui suivent toutes la durée du processus, de la spécification jusqu'aux tests de validation et d'intégration. Une des normes "mères" de la sûreté de fonctionnement des architectures électriques et électroniques est IEC-61508 [IEC 61508 2010]. Il s'agit d'un standard européen générique qui touche donc à de nombreux domaines tels que le ferroviaire, l'automobile, l'aéronautique, etc. Il s'agit en particulier des systèmes où il existe des risques pour les personnes ou sur l'environnement en cas de défaillances. Ce standard définit des niveaux de criticité SIL (*Safety Integrity Level*). Cela fourni des niveaux de fiabilité requis pour chaque niveau de criticité ainsi que les méthodes applicables pour atteindre cet objectif. Ce standard a été par la suite décliné suivant les domaines. Dans l'automobile est ainsi apparue en 2011 la première version d'ISO 26262, "Véhicules Routiers - Sécurité fonctionnelle [TC22/SC3/WG16 2011].

La norme ISO 26262 est la norme de référence pour la sûreté de fonctionnement dans le domaine automobile. Elle recommande des méthodes et mécanismes, applicables durant toutes les phases de développement du véhicule, pour atteindre et justifier son niveau de sûreté de fonctionnement. La norme préconise d'effectuer

une phase d'analyse des risques pour identifier les situations dangereuses et les classifier en 4 niveaux de criticités nommés ASIL (*Automotive Safety Integrity Level*) allant du moins critique (ASIL A) au plus critique (ASIL D). Pour la détermination de ces niveaux, trois critères sont pris en compte : la sévérité, la probabilité d'accomplissement et la contrôlabilité.

— **La Sévérité**
Les conséquences en cas de défaillance peuvent être Légères et Modérées (*S1*), Sévères et potentiellement mortelles – mais survie probable – (*S2*), Potentiellement mortelles -survie incertaine- voire mortelles (*S3*).

— **La Probabilité**
Le risque d'occurence peut être Très Faible (*E1*), Faible (*E2*), de probabilité Moyenne (*E3*) ou de Haute probabilité (*E4*).

— **La Contrôlabilité**
Les capacités de l'utilisateur à gérer la défaillance. Une défaillance peut être facilement contrôlable (*C1*), normalement contrôlable (*C2*), difficilement, voire impossible à contrôler (*C3*).

Bien entendu, l'interprétation de ces trois critères doit se faire au regard du système étudié, et non de façon absolue et déterministe. Ces critères donnent lieu à une table de classification qui définit alors le niveau d'ASIL des composants tel que décrit dans le Tableau 1.1. On remarquera le niveau "QM" pour *Quality Management*, qui correspond aux composants qui n'impliquent pas de criticité particulière et peuvent alors être développés "au mieux" sans contrainte spécifique. Il est bien entendu moins critique qu'un ASIL A. Avec cette table on connaît alors le niveau de confiance que l'on va imposer à chaque composant pour qu'il puisse être utilisé de façon sûre de fonctionnement. Plus un composant sera critique, plus son niveau d'ASIL sera élevé en conséquence, et donc plus il faudra apporter d'efforts pour qu'il respecte les standards.

TABLE 1.1 – Matrice de Définition des Niveaux d'ASIL - ISO 26262

		Contrôlabilité								
		C1			C2			C3		
	Sévérité	S1	S2	S3	S1	S2	S3	S1	S2	S3
Probabilité	E1	QM	QM	QM	QM	QM	QM	QM	QM	A
	E2	QM	QM	QM	QM	QM	A	QM	A	B
	E3	QM	QM	A	QM	A	B	A	B	C
	E4	QM	A	B	A	B	C	B	C	D

On peut mentionner de la même manière d'autres standards équivalents dans l'avionique, DO-176 où les niveaux de criticité sont désigné en DAL ("*Design Assurance Level*"), ou encore le ferroviaire avec CENELEC 5012x, mais aussi dans le nucléaire, le spatial, l'automatique, le médical... La plupart dérivés de IEC-61508. Un certain nombre de ces standards ont été comparés dans les travaux

de [Baufreton 2010]. On y retrouve un point commun qui nous intéresse tout particulièrement ici qui est sur les contraintes temporelles.

Que ce soit IEC-61508 ou plus spécifiquement ISO 26262, il est clairement stipulé que "*les contraintes temporelles des fonctions à durée critique doivent être gérées par les spécifications de sûreté de fonctionnement logicielle. Ici, à la fois les pires temps d'exécution au niveau du code et les temps de réponse au niveau système doivent être considérés.*". Et précisément, "*l'absence de tout interférence se doit d'être assuré et, tout comme dans IEC-61508, le logiciel est sujet au plus haut niveau d'ASIL impliqué quand l'indépendance temporelle entre les fonctionnalités ne peut être assurée.*" [2018]. C'est au regard de ce type de contrainte industrielle qu'il est essentiel de proposer de nouvelles solutions avec l'arrivée de calculateurs multicœurs qui complexifient grandement les garanties de non-interférence.

1.3.3 Contraintes d'intégration

De façon plus générale, les enjeux industriels peuvent varier selon les domaines. Ceci étant dit on peut nommer des points principaux, qui sont ceux que l'on va tenter de prendre en considération dans cette étude. La première d'entre elle est l'imposition de capacités de déploiement rapides ("*Time-to-market*" réduit). Les itérations entre générations demandent des coûts de développement les moins importants possibles. Cela permet des cycles courts et réactifs qui s'adaptent aux évolutions technologiques. Cette contrainte industrielle est structurante sur les choix de conception, ce qui nous ramène souvent au principe "*KISS*" pour "*Keep It Safe and Simple*" dans notre cas. C'est une philosophie que j'ai souhaité maintenir au long de cette thèse afin de tenter une approche un peu différente des principales recherches actuelles qui tentent souvent d'aller dans des niveaux de détails toujours plus précis et complexes pour répondre aux difficultés technologiques, au détriment d'une facilité d'implémentation qui permettrait une appropriation industrielle. Comme on le verra plus tard, il existe ainsi des solutions très sophistiquées qui donnent de bons résultats théoriques, mais qui ne se sont pas généralisés. Les questions de complexité d'implémentation et simplicité de maintenance dans un cas réel semblent donc relativement déterminantes pour mesurer la pertinence d'une nouvelle contribution à la sûreté des systèmes embarqués.

En ce sens, il existe globalement 3 types de sous-systèmes qui sont intégrés par les constructeurs. D'abord les systèmes en *black box* ou "boites noires" qui sont entièrement conçu par un équipementier tiers à partir de spécifications. Le contenu de ces "boites" est alors inconnu dans ses détails. Ensuite les systèmes en *white box* où à l'inverse, la totalité de sa conception et de son modèle est connu. Cela permet des tests bien plus en profondeur et donc une plus grande confiance en son comportement. Pour être des "boites blanches" les composants sont en général soit directement faits par l'intégrateur final, soit fourni en open-source (plus rare, tristement). Enfin, il y a tout l'entre-deux de composants contenants à la fois des éléments en "boite noire" et en "boite blanche", que l'on appelle naturellement "boites grises". Il est très important de noter par ailleurs une catégorie transverse des systèmes *"legacy"*, qui sont des composants où les altérations et modifications sont impossibles pour diverses raisons. Soit parce qu'il s'agit d'une black box que l'on

ne souhaite ou ne peut remplacer, soit pour des raisons de compatibilité restreinte où toute modification risquerait de compromettre la totalité du système intégré. Ces différents types de sous-systèmes sont importants à présenter, car tout ce qui va toucher aux boites noires et aux composants legacy ajoutent une contrainte forte sur le développement de solutions à la sûreté de fonctionnement. De fait, tout mécanisme de sûreté de fonctionnement impliquant la modification ou l'adaptation du code de fonctions intégrées au système risque d'être éliminé d'office !

On pourra pour finir, mentionner des enjeux plus divers tel que les contraintes d'encombrement. Les systèmes embarqués ont une forte tendance à la miniaturisation pour des raisons diverses selon les domaines. Cela permet une réduction de poids, essentiel pour tous les systèmes volants (avions, drones...) mais aussi d'encombrement pour des domaines comme l'automobile ou le ferroviaire qui doivent en toute circonstance rester dans des dimensions standards. Cette contrainte se fait beaucoup sentir avec l'arrivée des voitures autonomes par exemple, où les premiers prototypes – bien que fonctionnels – se sont avérés trop chargés et encombrants avec le surplus d'équipement pour être transposables facilement en produits commercialisables tel-quel.

Au regard de ces enjeux, l'évolution future naturelle est de réduire le nombre de calculateurs embarqués, en passant d'un grand nombre d'unités de calcul à une quantité limitée de "supercalculateurs", qui vont agréger différentes tâches. On passe de cette façon d'un système distribué à un système fédéré basé sur des calculateurs primaires accompagnés de processeurs satellites qui gèrent le strict nécessaire à hauteur des différents capteurs/actionneurs. Cela permet de réduire les coûts et l'encombrement, qui va diminuer par la même occasion la quantité de câblages requis. Ce type d'architecture va faciliter l'évolutivité qui sera donc bien plus axée sur des mises à jour logicielles sans toucher au matériel. La connectivité permet de mettre en œuvre le concept du véhicule *"as-a-service"*, qui va pouvoir évoluer et se mettre à jour régulièrement à distance (*Over-the-Air Updates*).

1.4 Grandes approches du domaine

Nous verrons plus en détail dans le chapitre 2 les différentes solutions actuelles qu'il existe dans le domaine pour répondre à ces problématiques. Nous pouvons tout de même d'ores-et-déjà présenter fondamentalement sur quoi reposent les principes existants afin de mieux situer notre axe de recherche.

La problématique principale à laquelle nous devons faire face réside dans la gestion des interférences matérielles de façon à éviter des fautes temporelles ou tout du moins à couper la chaîne de causalité de façon à ce qu'il y ait toujours silence sur défaillance et donc que le système puisse continuer à fonctionner correctement. Il est possible de différencier deux grands domaines d'approches. D'une part les stratégies de **contrôle** qui sont plutôt statiques et définis hors-ligne lors du développement ; d'autre part les stratégies **réactives** qui sont plutôt dynamiques et évoluent en ligne pendant le fonctionnement.

1.4.1 Mécanismes de contrôle

Ce genre de stratégies consistent à déployer des mécanismes qui limitent les interférences et donc les risques de faute de façon préventive. Le développement et l'implémentation sont alors réalisés d'une manière à ce que par construction, les risquent soit *de facto* rendus impossibles. Ce type de stratégies permet de limiter plus efficacement les explosions de pire temps d'exécution notamment, et donc conserver une exécution du logiciel bien cadrée et maîtrisée pour en contrôler les risques inhérents au matériel.

On peut citer parmi ce genre de techniques :

— Politiques strictes de gestion d'accès aux ressources partagée, avec des intervalles de temps fixes dédiées notamment. Chaque application ayant sa fenêtre temporelle dédiée pour accéder à la ressource partagée, les délais deviennent connus et maîtrisés.

— Séparation temporelle d'exécution des applications : il est possible d'ordonnancer les différentes applications de façon complètement séparées les unes des autres. Cela revient à limiter fortement l'exécution parallèle de code, mais par la même prévient radicalement tout risque d'interférence avec le logiciel ainsi isolé.

— Séparation spatiale des applications : l'allocation d'un espace mémoire dédié pour chaque application permet de réserver et donc séparer physiquement les applications entre-elles. De cette façon, tout risque de recouvrement des données (c.f. erreurs de lecture) est empêché entre applications qui ont des réservations d'espace mémoire disjoint.

La plupart des méthodes qui rentrent dans cette catégorie ont en revanche un défaut commun qui est de limiter les capacités d'utilisation du matériel. En effet, on comprend naturellement que si l'on limite la taille mémoire qu'une application donnée est autorisée à utiliser pour son fonctionnement, ou encore si l'on contraint sa plage temporelle d'accès à certaines ressources alors les performances de l'ensemble seront forcément moindre que s'il n'y avait pas ces limitations. Des garanties réduites sur les pires temps d'exécution se font donc au détriment de l'optimisation d'utilisation des ressources matérielles.

Si l'on souhaitait se passer de tels mécanismes en conservant les mêmes niveaux de certitudes sur les durées d'exécution des tâches, cela impliquerait un surdimensionnement non négligeable des processeurs utilisés.

1.4.2 Mécanismes Réactifs

À l'inverse, les stratégies réactives se basent sur l'observation de l'état du système pendant son fonctionnement de façon à agir en conséquence uniquement si nécessaire. Le principe est de monitorer en temps-réel l'exécution des processus et activer sur demande des mécanismes de prévention des fautes. Ce type de méthode est plus complexe à mettre en place et présentent a priori des garanties plus faibles. Cela en fait une solution plus adaptée pour du temps-réel souple tout en conservant des performances moyennes convenables.

Les systèmes de **watchdog** sont à la base de ce genre de mécanisme, en levant un traitement d'erreur en cas de constat d'une défaillance, de façon à isoler cette dernière. Cela ne permet pas d'empêcher l'erreur, uniquement de prévenir ou au moins mitiger toute conséquence supplémentaire.

Les techniques d'ordonnancement dynamique des tâches permettent aussi en un sens d'optimiser l'utilisation des ressources de calcul en priorisant l'exécution au plus urgent par exemple. C'est le cas par exemple d'un ordonnancement des tâches en EDF - "*Earliest Deadline First*", autrement dit "Priorité à l'Échéance au plus Tôt" qui exécute, comme son nom l'indique, systématiquement la tâche dont l'échéance est la plus proche. De façon générale, tout mécanisme de changement dynamique de priorité selon l'état du système entre dans cette catégorie.

Enfin, les mécanismes de réaction consistent essentiellement à suspendre toute tâche indésirable de façon à isoler les tâches temps-réel en cours d'exécution. Par conséquent, cela prévient pendant l'exécution les risques d'interférences matérielles, au détriment temporaire des tâches non critiques. Cela n'est bien entendu possible que dans un cadre où l'on peut modifier en fonctionnement l'exécution des tâches et se permettre d'en stopper une partie. Certains mécanismes réactifs sont à usage unique dans le sens où une fois ce déclenchement fait, le système reste dans un fonctionnement en mode dégradé pour tout le reste de son exécution. À l'inverse d'autres solutions proposent un mode dégradé temporaire avec un retour en fonctionnement nominal une fois le risque passé.

La plus grande difficulté de ces stratégies réside donc dans la preuve des garanties qu'elles sont capables de fournir sur les propriétés de sûreté de fonctionnement au regard des exigences non fonctionnelles définies. Elles permettent de mieux exploiter les ressources disponibles. Les systèmes réactifs sont par exemple à la base des usages informatiques grand public qui reposent sur un système d'exploitation standard pour lesquels il n'y a pas de contrainte temporelle dure.

1.5 Contribution de la thèse et objectifs

Dans le cadre de l'utilisation de calculateurs multicœurs dans les applications industrielles, nous aborderont dans cette thèse les difficultés que cela implique en terme d'implémentation pour continuer à garantir le bon fonctionnement du logiciel. Plus spécifiquement, nous nous intéresseront aux implications du partage de ressources sur les temps d'exécution de logiciel critiques dans le cadre de systèmes à criticité mixte. Ces partages pouvant entraîner des congestions qui en conséquence ajoutent des latences qui peuvent aller jusqu'à provoquer des dépassements d'échéance temporelle et donc des fautes logicielles temporelles transitoires. Pour empêcher ce risque potentiellement critique, il sera proposé un mécanisme novateur de Surveillance et de Contrôle d'exécution logiciel. Les objectifs ici sont multiples, car d'une part, l'on souhaite conserver des garanties sur les temps d'exécution de logiciel critique, mais en même temps il faut trouver des mécanismes les moins intrusifs possibles sur l'exécution pour profiter au maximum des puissances de calcul multicœur mis à disposition.

Pour cela, nous verrons dans le chapitre 2 sur l'État de l'Art les différents

propositions existantes qui permettent de répondre à tout ou partie des objectifs susmentionnés.

Par la suite le chapitre 3 présentera notre façon d'aborder la question avec nos hypothèses et modélisation du système. Dans le cadre d'un système multicœur qui héberge des applications à criticité mixte, on verra le modèle d'exécution adopté, orienté vers une approche originale basé sur des chaînes de tâches. Cela va impliquer des notions de précédence d'exécution et de temps de réponse bout-à-bout qui seront essentiels par la suite.

Cela nous mènera dans le chapitre 4 suivant à développer notre mécanisme de Surveillance et Contrôle basé sur ces chaînes, son architecture et ce que cela implique en terme d'implémentation. Cette approche se basera sur la surveillance des contraintes temporelles de chaînes de tâches dans un système à criticité duale, exécuté sur un multicœur bien entendu. L'objectif étant de stopper temporairement des tâches non critiques pour éviter des interférences qui risqueraient de provoquer des fautes temporelles.

Enfin, nous verrons en chapitre 5 un cas d'implémentation qui a pu être réalisé sur une plateforme d'essai. Cette plateforme se voulant être une preuve de concept, l'objectif est de voir l'influence et les tenants et aboutissants du mécanisme proposé. Nous utiliserons pour cela des tâches d'une suite de benchmark sur laquelle on pourra implémenter le mécanisme, le calibrer et en tirer des mesures de performance.

En conclusion, nous ferons un bilan des résultats obtenus avec des perspectives d'utilisation, ainsi que des pistes de recherche.

Chapitre 2

Etat de l'Art

Sommaire

2.1 Optimisation des ressources CPU

2.1.1 Allocation des tâches - optimisation d'ordonnancement

2.1.1.1 Fair scheduling - OS General Purpose

2.1.1.2 Systèmes à criticité mixte

ordonnancements statiques partitionnements spatio-temporels exemple Hyperviseur PikeOS Automotive case : AUTOSAR timing constraints Avionic Case : ARINC653

2.1.2 Autres considérations

en général on va rarement optimiser à 100%, mais du coup exploiter d'autres critères comme la consommation d'énergie, la chauffe etc... Des critères d'isolation des tâches pour des raison de sécurité peuvent aussi être faits... on abordera pas plus que cela ces éléments dans cette thèse.

2.1.3 Limitations et systèmes plus réalistes

2.1.3.1 Limitations des solutions actuelles

impossibilité de totalement simuler et maîtriser de façon prédictive le comportement et donc les risques de fautes liées aux interférences : système trop complexe.

2.1.3.2 Systèmes à modes dégradés et améliorations

Disponibilité des tâches à criticité basse. Diminution de priorités. Élongation des tâches à criticité basses et contrôle de budget. sur la mémoire, [Blin 2017] typiquement sur l'ordonnancement Migration de tâches.

Chapitre 3

Principe et architecture

Sommaire

Dans ce chapitre, nous allons regarder plus en détail les hypothèses que nous avons considérées pour proposer un mécanisme de Surveillance et de Contrôle. Le contexte industriel mentionné précédemment au chapitre 1 a grandement contribué aux spécifications de notre proposition. De fait et pour rappel, les objectifs principaux de ces travaux est de proposer un mécanisme logiciel qui permette dans le même temps à utiliser au maximum les ressources matérielles disponibles et avoir des garanties minimales sur les temps d'exécutions. Tout l'enjeu de cette proposition est d'arriver à une solution qui limite les coûts de développement notamment en étant compatible avec des composants logiciels *legacy* et/ou en boîte noire qui ne permettent pas d'être modifié dans leur fonctionnement interne. Mais aussi en limitant les besoins de modification de l'architecture logicielle suite à des mises à jour ou l'ajout de fonctionnalités supplémentaires. Par ailleurs, on tentera de s'abstraire le plus possible du contexte automobile pour proposer une approche généraliste qui puisse s'adapter à différents domaines et selon le contexte.

Nous verrons au sein de ce chapitre dans un premier temps les prérequis considérés ainsi que notre approche sur la modélisation du problème. Suite à cela nous verront dans un second temps la solution proposée qui est un mécanisme de sûreté de fonctionnement réactif de type Surveillance et Contrôle. Son objectif sera de prévenir les fautes temporelles par dépassement d'échéances sur les tâches critiques, via une approche basée sur des chaînes de tâches. Pour finir, nous verrons dans quelle mesure notre proposition se positionne vis-à-vis des standards d'architectures existants dans différents domaines industriels dont l'automobile.

3.1 Un modèle basé sur des chaînes de tâches pour garantir les contraintes temporelles

Je sais ce que j'ai loupé d'entrée de jeu !

Il faut que je rajoute l'explication de la philosophie générale, i.e. taches critiques interférées par non critiques, passage en mode dégradé pour que les taches critiques se retrouvent en pseudo isolation sans interférences.

Afin d'étudier et développer notre mécanisme de gestion de fautes temporelles dans le cadre d'un système à criticité mixte (*Mixed Criticality Systems*), nous avons besoin de formaliser la représentation des tâches qui seront à l'étude et leur modèle. Remarquons que le modèle ici proposé est relativement arbitraire et choisi essentiellement pour des raisons de commodité. De fait, on retiendra deux critères principaux pour guider le choix de notre modèle : la simplicité d'implémentation et l'accessibilité à des suites logicielles qui peuvent servir de tâches pour simuler un système réel lors de nos tests. L'objectif est ainsi de trouver un juste milieu entre un modèle représentatif d'une réalité technique dans les milieux industriels d'une part et un modèle qui nous évite des surcoûts de développement pour obtenir une première preuve de concept fonctionnelle.

Ce modèle doit décrire d'une part la méthode d'exécution des tâches, la façon d'interagir, entre-elles, notamment pour les tâches à haut niveau de criticité qui sont reliées sous la forme d'une chaîne pour réaliser une fonction critique. Il est à noter que le mécanisme de sûreté de fonctionnement que nous proposons par la suite est *in fine* indépendant du modèle de tâche ici proposé. Il conviendra d'adapter au besoin la partie de Contrôle du mécanisme, de façon à ce qu'elle prenne en compte l'état d'exécution du système selon le modèle de tâche utilisé, s'il est différent de celui présenté ici. Typiquement la vérification des contraintes de précédence peut différer. On aura l'occasion d'aborder rapidement ces aspects par la suite, avec quelques exemples de modifications requises suivant des changements de ce modèle de tâche. *Petite note pour pas oublier*

3.1.1 Notion de Criticité

Les systèmes à criticité mixte ont mené à de nombreuses études pour gérer la répartition temporelle des tâches, et en même temps prendre en compte les conditions de partitionnement et le partage des ressources de façon à optimiser l'usage de ces dernières. La plupart des travaux dans ce domaine se basent sur le modèle d'un ensemble de tâches τ_i caractérisées par (T_i, D_i, C_i, L_i) :

— T_i - la période d'apparition de la tâche

— D_i - la date limite d'échéance d'exécution (que l'on appelle communément *deadline*)

— C_i - Le Temps d'Exécution Pire Cas

— L_i - Le Niveau de Criticité de la tâche

Avant toute chose, il est à noter que la notion de criticité des tâches que nous avons déjà mentionnée dès le chapitre d'introduction est polysémique. En effet, selon

le domaine et la spécialisation des interlocuteurs, ce mot peut traduire des enjeux de sûreté de fonctionnement différents. Il convient donc de clarifier cette notion de criticité pour notre cas avant d'aller de l'avant. Ce problème de sens a été souligné notamment par [Graydon 2013]. Il existe deux grandes approches à la notion de criticité.

La première, d'un point de vue modélisation de modèle d'exécution temps-réel, se retrouve essentiellement dans les recherches académiques. La notion de criticité est reliée aux contraintes temporelles imposées sur les tâches. Cela se traduit par le niveau de fiabilité de l'estimation des pires temps d'exécution C_i [Vestal 2007]. Plus le niveau de criticité sera élevé, plus ce WCET sera estimé de façon fiable (l'on pourrait dire de façon "prudente") et avec des contraintes fortes. Autrement dit, un haut niveau de criticité requiert de prendre en compte les cas les plus extrêmes de retard d'exécution pour estimer le WCET et les temps d'exécutions estimés pour l'ordonnançabilité sont plus longs. Ce type de formulation a été étendue avec des modes de criticité différente. À chaque mode de criticité est associé un niveau de criticité des tâches, plus ou moins pessimiste. Tout l'intérêt de ces modèles dans le cadre de systèmes à criticité mixte est de proposer des stratégies d'ordonnancement qui prennent en compte ces niveaux et ces modes de criticité pour permettre la priorisation d'exécution de certaines tâches plus critiques que d'autres, pour aller jusqu'à stopper des tâches moins critiques pour garantir l'exécution des tâches de criticité supérieure. Cette façon d'aborder la criticité est donc totalement orientée suivant des critères d'ordonnancement et de respect des échéances temporelles.

On notera dès cette première formulation un mélange entre les niveaux de criticité des tâches et des modes de criticité.

Le second, d'un point de vue des standards de sûreté de fonctionnement, est largement exploité dans l'industrie. Dans ce cadre-là, la criticité détermine un niveau de confiance que l'on peut accorder à un composant tel que décrit précédemment en sous-section 1.2.1. Le niveau de criticité est alors déterminé non pas au regard des contraintes temporelles, mais plutôt avec des analyses de sûreté (Analyses de Risques, FMEA, FTA...) pour obtenir une catégorisation tel que défini par les différentes normes du domaine (un niveau d'ASIL tel que défini dans ISO26262 pour l'automobile par exemple). Elle se définit alors en fonction de critères comme **a)** l'évaluation des conséquences en cas de défaillance, **b)** la probabilité de défaillance, **c)** les moyens disponibles pour compenser ou gérer la faute si elle survient. Par conséquent, le niveau de criticité d'une application ne reflète alors pas nécessairement la sévérité ou les répercutions d'une faute. De fait, une application avec des chances de défaillance très faibles, mais où les répercussions sont très graves pourra être d'un niveau de criticité faible. Inversement, une application avec peu, voire aucune, conséquence en cas de faute mais un risque de défaillance très élevé peut avoir le même niveau de criticité. Avec ce type d'interprétation, le principe d'arrêt de tâches à niveau de criticité inférieur pour éviter les fautes temporelles sur des tâches plus critiques perds de son sens. Alors même que ce genre de solution est courant dans les mécanismes de sûreté de fonctionnement de systèmes à criticité mixte, mais avec la première définition suscitée de la criticité. De plus, les niveaux de criticité dans le domaine industriel impliquent souvent des propriétés de composabilité, qui permettent de combiner des composants à plus faible niveau de criticité pour obtenir

l'équivalent d'un unique composant de niveau de criticité supérieure. Par exemple, si un capteur automobile requiert pour son usage un niveau d'ASIL A, il est possible d'utiliser conjointement deux capteurs avec un niveau d'ASIL B en équivalent. L'objectif des compositions étant de limiter les coûts, étant donné que plus le niveau d'ASIL est élevé, plus les exigences de développement et de vérification sont forts.

Enfin, que ce soit dans un cas comme dans l'autre, il est question de **Modes** de fonctionnement. Mais là encore cela peut impliquer des sens différents. Il existe d'une part les **Modes de Service**, c'est-à-dire des modes de fonctionnement dont l'objectif est uniquement la reconfiguration pour la survie et le bon fonctionnement du système. Il y a ensuite les **Modes d'Opération** qui décrivent des modes de fonctionnement dans l'usage "normal" du système. Par exemple des modes "décollage", "vol de croisière" et atterrissage" pour un avion. Et enfin les **Modes d'ordonnancement** qui se focalise exclusivement sur la façon selon laquelle le processeur va exécuter (ou non) les tâches pour permettre le bon ordonnancement global du logiciel.

Au regard de ces disparités, il convient donc de spécifier selon quel critère nous souhaitons définir les tâches *critiques* pour lesquelles nous désirons conserver des garanties de temps d'exécution pire cas. L'objectif inclusif de ces travaux étant de limiter pour une application donnée les risque de dépassement d'échéances d'exécution du logiciel qui réalise le service essentiel du système, les définitions se veulent volontairement générales pour pouvoir s'adapter selon les besoins du cas d'étude.

Définition 3.1.1. – Criticité
La criticité d'une tâche exécutée sur un processeur donné pour réaliser une fonctionnalité du-dit processeur se définit selon son **importante**. L'importance se mesure par les conséquences en cas de défaillance de cette fonctionnalité à la fois au regard de l'utilisateur (dangers) ainsi que l'importance de la fonctionnalité vis-à-vis des cas d'application essentiels du système.

Le système que nous allons étudier ici est dit à niveau de criticité dual. Il exécute un set de tâches logicielles (aussi appelée "charge utile") exécutées sur un support logiciel (classiquement, le système d'exploitation). Elles se répartissent entre les tâches à haute criticité d'une part ("tâches critiques"), et à faible criticité d'autre part (non critiques). Dans ce cadre particulier, il est alors possible de définir :

Définition 3.1.2. – Tâche critique
Une tâche critique est une tâche au niveau d'importance élevée. On peut aussi parler communément de tâche "vitale", dans le sens où une défaillance de cette fonctionnalité aurait soit des conséquences sévères pour l'utilisateur, soit empêcherait le bon fonctionnement d'un des cas d'application essentiels du système. Inversement, une tâche pour laquelle une faute provoquant un dépassement d'échéance n'aurait pas de conséquences sévères sur l'utilisateur ou ne préviendrait pas la réalisation des fonctionnalités essentielles peut se définir comme non critique. En d'autres termes, une tâche de faible importance (relativement aux tâches critiques).

Enfin, en cohérence avec les modes de criticité mentionnés plus haut, nous définissons deux Modes de Service directement reliés à des Modes d'Ordonnancement pour contribuer à la sûreté de fonctionnement du système. D'une part le **Mode Nominal** où tout le système fonctionne normalement en l'absence de fautes. D'autre part, le

Mode Dégradé dans lequel le système ne réalise pas la totalité des fonctionnalités de façon à pouvoir conserver des garanties d'exécution sur les tâches critiques et donc prévenir les fautes.

3.1.2 Modèle de Tâches et Chaînes de tâches

Maintenant que nous avons clairement posé la notion de criticité sur laquelle nous nous focalisons, il est possible d'expliciter notre objectif pour répondre à la problématique.

Pour rappel, nous souhaitons exploiter au maximum les ressources matérielles disponibles au sein d'un multicœur basé sur le cache. Dans le même temps, la contrainte propre aux fonctionnalités critiques du système nous impose de conserver des garanties de sûreté de fonctionnement de façon à éviter toute défaillance catastrophique. Pour répondre à cela, nous nous plaçons alors dans le cadre d'un système où les tâches sont divisées entre deux niveaux de criticité. D'une part les tâches non critiques qui sont potentiellement moins restreintes en terme d'usage de ressources (temps de calcul et ressources partagées). D'autre part les tâches critiques suivant le critère d'importance susmentionné pour lesquelles on doit garantir une qualité de service.

Insérer petit schéma intermédiaire avec des tâches verts et rouges (critiques/non critiques) sur un multicœur ?

3.1.2.1 Modèle de tâches

La plupart des hypothèses faites ici se focalisent sur les tâches critiques, tandis que la seule hypothèse forte sur les tâches non critique est la capacité à les stopper (soit un arrêt total, soit une mise en pause) et les relancer en cours d'exécution de façon à pouvoir déclencher un Mode Dégradé où il n'y a plus de tâches non critiques avec les risques d'interférences afférents envers les tâches critiques. Sous les systèmes type Unix, cela correspond typiquement à l'envoi d'un signal SIGSTOP et SIGCONT. Sans cette condition, les Modes de Service mentionnés ci-dessus ne sont pas exploitables pour notre besoin.

Chaque tâche critique τ_i est activée et exécutée suivant une période T_i. À chaque période, le job $\tau_{i,j}$ correspond à la j^{ieme} exécution de la tâche τ_i. On peut alors noter pour chaque job $\tau_{i,j}$ son moment d'activation $a_{i,j}$, son début d'exécution $s_{i,j}$ et sa terminaison $e_{i,j}$. On considère qu'un job consomme toutes ses données d'entrée (inputs) au début de son exécution, s'exécute et fourni à la fin de son exécution les données de sortie. Les données d'entrée et de sortie des tâches sont stockées en espace mémoire partagé : la transmission des données d'une tâche à l'autre se fait de façon asynchrone. Cela nous mène à la question de l'interaction entre les tâches et notamment la façon de représenter la précédence.

Citer différents modèles d'exécution de tâches existants ici ? c.f. [Friese 2018]

3.1.2.2 Chaînes de tâches

La question de la dépendance entre les tâches est importante pour aborder le problème des contraintes temps-réel avec une vision plus macroscopique. En effet,

dans le cadre de l'usage de tâches ayant des contraintes temporelles souples (c.f. section 1.2.2 - Systèmes temps-réel), c'est uniquement avec une vision plus globale de l'exécution du système qu'il est possible de tirer au maximum parti des légers dépassements pour éviter dans la globalité d'avoir recours à des politiques d'exécution des tâches plus restrictives, et par conséquent qui sous-exploitent la puissance de calcul disponible. Nous considérons ici la dépendance entre les tâches via les données partagées entre ces dernières selon un modèle type producteurs/consommateurs. Les tâches ont des relations de cause à effet et par conséquent, d'un point de vue strictement fonctionnel on peut décrire le système comme étant une accumulation de fonctionnalités réalisées par l'exécution de tâches successives. Cela permet alors d'introduire la notion de contrainte temporelle fonctionnelle, qui décrivent des contraintes d'exécution de chaînes de tâches bout-en-bout.

On représente une dépendance entre tâches sous la forme de chaînes de tâches, suivant le modèle $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$. Dans un tel exemple, τ_1 est la **tâche d'entrée** de la chaîne, tandis que τ_n est la **tâche de sortie** de la chaîne. Notons que ce modèle peut être étendu pour supporter des tâches représentées par un Diagramme Orienté Acyclique (*Directed Acyclic Graph - DAG*) sans difficulté. Nous nous contentons ici de travailler avec des chaînes directes, sans divergences ou convergences dans le graphe. Nous aborderons la question ultérieurement. De fait, cet ajout de complexité dans le modèle de chaîne de tâche n'apporte rien sur les résultats ni sur la démarche et n'implique, au demeurant, pas de modifications sur la solution proposée.

J'hésite à présenter ça dans "l'autre sens" : présenter un modèle de chaînes de tâches plus complet (avec divergences, convergences, etc.) et au final restreindre le modèle à des chaînes linéaires qu'au niveau du cas d'étude (chapitre 5). Option 2 (actuelle) : en perspective de la thèse présenter les implications d'un modèle de chaînes plus complexe

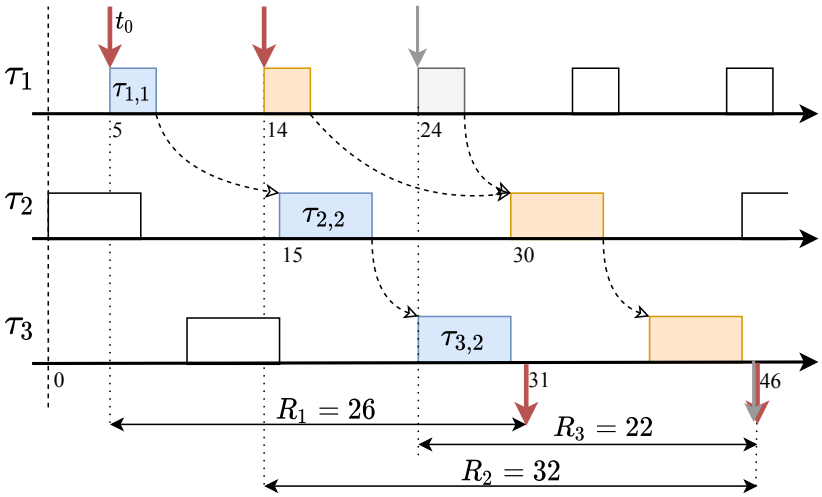


FIGURE 3.1 – Exemple d'exécution d'une chaîne de tâches $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$

Dans ce contexte, on peut donc définir la relation entre une tâche τ_i et son

successeur τ_{i+1}. Pour produire la donnée de sortie du job $\tau_{i+1,k}$ de la tâche τ_{i+1}, ce dernier consommes toutes les données d'entrée en attente provenant des jobs $\tau_{i,j}$. Les données en attente étant celles qui n'ont pas été consommé par le job précédent de τ_{i+1}, i.e. $\tau_{i+1,k-1}$. On peut donc écrire que pour un $\{i,k\}$ donnés, sont consommés les données de tous les jobs $\tau_{i,j}$ ssi j tel que $e_{i,j} \leq s_{i+1,k}$ et $e_{i,j} > s_{i+1,k-1}$. Autrement dit, un job $\tau_{i,j}$ n'a un effet sur $\tau_{i+1,k}$ si et seulement si ce dernier est le premier job de τ_{i+1} exécuté après la terminaison de $\tau_{i,j}$.

Dans ces conditions, on nomme $\tau_{i+1,k}$ le **successeur** du job $\tau_{i,j}$. On note $succ()$ la fonction qui permet de trouver le successeur d'un job donné. Par extension, la fonction itérative $succ^{n-1}()$ permet de trouver le job de sortie d'une chaîne de tâche donnée, selon le job d'entrée. Pour illustrer cela, on peut prendre l'exemple d'une chaîne de trois tâches $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, tel que représenté en Figure 3.1. On peut par exemple voir qu'une des exécutions de la chaîne de tâche, débutant par $\tau_{1,1}$, donne : $succ^2(\tau_{1,1}) = succ(succ(\tau_{1,1})) = succ(\tau_{2,2}) = \tau_{3,2}$. Mais aussi que le job $\tau_{2,3}$ doit prendre en compte les valeurs de sorties de 2 jobs de la tâche τ_1.

Étant donné que les tâches peuvent être définies par des périodes d'activation différentes, cela signifie notamment que si une tâche τ_i est exécutée plus fréquemment que son successeur $succ(\tau_{i+1})$, alors il est possible qu'un job $\tau_{i+1,j}$ soit le successeur de plusieurs jobs de la tâche τ_i. Cette façon de considérer les choses permet une plus grande flexibilité de notre modèle pour s'adapter à un cas concret. En effet, de cette façon le modèle gère déjà un nombre assez significatif d'implémentations de tâches existantes :

— les tâches qui ne considèrent que la donnée d'entrée la plus récente, les données précédentes étant considérées obsolètes. Dans ce cas-là, si les données de 2 jobs prédécesseurs sont consommés, en réalité la donnée du job le plus ancien des deux sera simplement ignorée.

— Les tâches avec une file d'attente en entrée : lorsque la tâche est exécutée elle consomme toutes les données d'entrée en attente.

— Pour les tâches où chaque exécution de la tâche ne prend en compte qu'une seule donnée de la file d'attente, ce n'est géré que partiellement. Cela peut être la donnée la plus ancienne (stratégie FIFO [1]) ou la plus récente (stratégie LIFO [2]). Si une tâche s'exécute plus lentement que sa prédécesseure, c'est le cas que nous ne gérons pas. Cependant, il est improbable, car cela implique que la quantité de données en file d'attente peut potentiellement exploser (or, la file d'attente ne peut être infinie). En revanche si les tâches ont toutes la même fréquence d'activation, ou si les tâches prédécesseures s'exécutent systématiquement à une plus faible fréquence que les tâches suivantes, alors on se retrouve finalement dans la même situation que le premier type de tâches citées. De fait cela implique qu'il n'y a en réalité jamais plusieurs données en attente à l'entrée en fonctionnement normal.

De façon succincte, quand on parle de "consommer" plusieurs jobs de la tâche précédente tel qu'on le présente ici, cela n'implique pas que toutes ces données seront

1. FIFO : First-In First-Out, les données sont traitées de la première arrivée à la dernière.
2. LIFO : Last-In First-Out, les données sont traitées de la dernière (plus récente) arrivée à la plus ancienne.

prises en compte. Tout dépend du modèle de fonctionnement interne des tâches. Mais quoi qu'il en soit, cela permet de gérer un grand nombre de comportements classiques.

Ce modèle présente en revanche un inconvénient qui est de ne pas considérer de potentiels retards entre le moment où une tâche délivre sa donnée de sortie et le moment où cette donnée est réellement disponible pour son successeur. Il faudrait pour cela ajouter une constante de latence correctement estimée selon le système à la condition de précédence de tâche.

Temps de réponse bout-en-bout La notion de successeur permet de définir le temps de réponse bout-en-bout R_j de la $j^{ème}$ instance d'exécution d'une chaîne de tâche. Ainsi R_j désigne le temps écoulé entre l'activation du *job* d'entrée $\tau_{1,j}$ de la chaîne, jusqu'à la terminaison du *job* de sortie $\tau_{n,k} = succ^{n-1}(\tau_{1,j})$. On a alors $R_j = e_{n,k} - a_{1,j}$ que l'on peut retrouver dans l'exemple du Figure 3.1. Sur cet exemple, il est possible de reconstituer trois instances d'exécution de la chaîne de tâche avec les trois temps de réponse correspondants : R_1, R_2, R_3.

Intuitivement, l'échéance bout-en-bout représente alors la durée maximale acceptable pour qu'une donnée d'entrée de la chaîne ait un effet du côté de la sortie. Pour une fonctionnalité donnée on comprend bien que cette échéance doit être bornée, et qu'il faut donc des garanties pour que tout se passe bien bout-en-bout. À considérer l'échéance d'une chaîne de tâche D, pour éviter toute faute temporelle de non-respect d'échéance, il faut a minima respecter : $\max_{j\in\mathbb{N}}\{R_j\} \leq D$.

L'objectif à présent est de proposer une approche qui permette justement d'exploiter ces contraintes bout-en-bout, de façon à éviter les risques de fautes temporelles au niveau fonctionnel. Cela se traduit par la volonté de prévenir les risques de dépassement d'échéances, non pas au niveau de chaque tâche individuelle mais plutôt à un niveau d'observation au-dessus qui est en lien direct avec la représentation fonctionnelle.

3.2 Mécanisme d'anticipation par Surveillance et Contrôle

3.2.1 Méthode d'anticipation

Je propose donc un mécanisme basé sur la surveillance à l'exécution de l'avancement d'une chaîne de tâche. Pour ce faire, on introduit les notions d'**État de Chaîne de Tâche** et de **Trace d'Exécution de Chaîne de Tâche**. Une chaîne de tâche donnée est associée à un État et plusieurs Traces d'Exécutions. Ces deux éléments évoluant au fil de l'exécution du système, on peut noter $S_C(t)$ l'État et $ET_C(i,t)$ une Trace d'Exécution donnée de la Chaîne de tâches.

On peut alors définir pour une Chaîne de Tâche dont la tâche d'entrée est τ_1, et la tâche de sortie τ_n :

Définition 3.2.1. Une **Trace d'Exécution** $ET_C(i)$ se définit par un job d'entrée ainsi que tous les *successeurs* itératifs de ce job.

$$S_C(i) = \{\tau_{1,i}\ ,\ succ(\tau_{1,i})\ ,\ \dots\ ,\ succ^n(\tau_{1,i}) = \tau_{n,i}\}$$

À un instant t, une Trace d'Exécution est dite *active* si son job de départ $\tau_{1,i}$ a déjà été activé, et que son successeur itératif correspondant à la tâche de sortie de la chaîne $\tau_{n,i}$ n'a pas encore été terminé. Autrement, elle est *inactive*. En d'autres termes :

$$TE_C(i,t) \quad \text{est active ssi} \quad a_{1,i} \leq t \quad \text{et} \quad e_{n,i} > t$$

À un moment t de l'exécution du système, il est possible de définir à partir de l'état des Traces d'exécution (actives ou inactives) l'État de la Chaîne de tâches :

Définition 3.2.2 (Etat d'une Chaîne de Tâche)**.** L'État d'une Chaîne de Tâche définie à un instant t l'état d'avancement de l'exécution de la chaîne de tâche qui est toujours en cours et a été activée la plus anciennement. En d'autres termes :

$$S(t) = \langle t_0, \tau_i \rangle$$

Avec t_0 la plus ancienne activation parmi les $ET_C(i,t)$ et τ_i la prochaine tâche de cette trace d'exécution qui n'a pas encore été exécutée.

De cette façon, l'État d'une chaîne de tâches indique quelles sont les tâches restantes à exécuter dans la chaîne à un instant donné et son temps de réponse partiel actuel que l'on notera $RT(t) = t - t_0$.

Pour finir, au regard de l'État d'une chaîne de tâches, on peut s'intéresser au temps restant à la complétion de cette chaîne. Il est possible d'estimer le temps qu'il faudra pour aller jusqu'à exécuter la tâche de sortie de la Trace d'Exécution active observée. Et si de plus cette estimation est faite dans la même logique qu'une estimation de Temps d'Exécution Pire Cas, on obtient alors une estimation de Pire Temps de Réponse restant $rWCRT(t)$ à l'instant t.

Alors, en combinant l'État à un instant t avec une estimation du Temps de Réponse Pire Cas restant, on peut donc estimer une borne haute garantie de temps de réponse de la chaîne de tâche. C'est là que l'on peut faire entrer en jeu un mécanisme d'anticipation. On dispose pour une chaîne de tâches donnée de son temps de réponse partiel actuel ainsi qu'une estimation de temps de réponse restant en pire cas. Il est alors possible de déterminer s'il y a un risque de défaillance par dépassement de l'échéance bout-en-bout D.

Théorème 3.2.3 (Risque de dépassement d'échéance)**.** *Si à un instant donné t, l'inéquation suivante est respectée, alors il y a risque de dépassement d'échéance.*

$$RT(t) + rWCRT(t) \geq D_C$$

Pour illustrer cette logique, on peut voir sur le chronogramme 3.2 à nouveau un exemple avec une chaîne de tâche $\tau_1 \to \tau_2 \to \tau_3$. À l'instant $t = 18$ indiqué il y a deux Traces d'Exécution actives (chaînes reliées par une flèche de succession). On a de ce constat l'État de la chaîne $S_C(t) = \langle \tau_0, \tau_2 \rangle = \langle 5, \tau_2 \rangle$. On en déduit

$RT(18) = t - t_0 = 18 - 4 = 14$. Si l'on ajoute à cela une estimation du Temps de Réponse Pire Cas restant $rWCRT(\tau_i)$, qui est le temps estimé pour que $\rightarrow \tau_2 \rightarrow \tau_3$ soit exécuté selon les contraintes de précédence, alors on a l'estimation du Pire Temps de Réponse : $RT(18) + rWCRT(18) = 33$ que l'on peut comparer à la date d'échéance $D_c = 30$. Dans cet exemple, il existe donc un risque de dépassement de l'échéance. Il est à noter aussi que dans cet exemple l'instant t a été pris en plein pendant l'exécution de la tâche τ_2. Ce qui est pris en considération comme si cette dernière n'était pas exécutée. Si l'on ne prend pas non plus en compte son exécution partielle, c'est parce que d'un point de vue externe à cette tâche, sans l'instrumentaliser il n'est pas possible de le savoir. Et de fait, l'une de nos contraintes étant d'être le moins intrusif possible sur le code, notamment pour les cas où certains logiciels ne sont pas modifiables (black-box ou legacy).

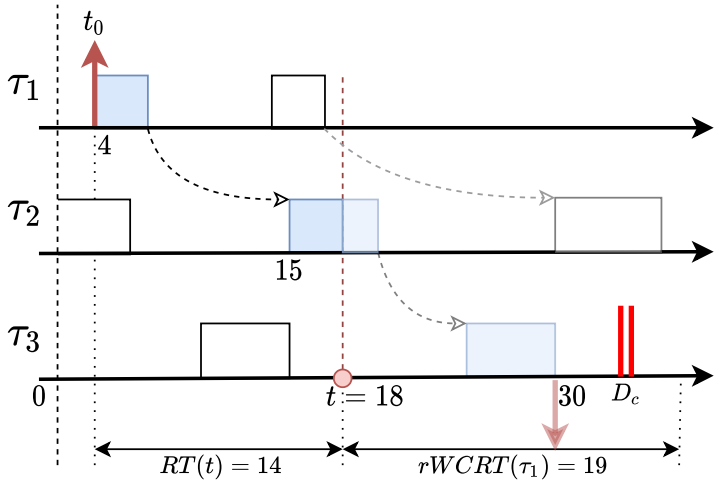


FIGURE 3.2 – Exemple de trace d'exécution et Etat de la Chaîne pour anticipation

À présent, il faut nous souvenir de notre objectif dans tout cela. Pourquoi vouloir anticiper une défaillance de la sorte ? Pour rappel l'objectif est d'anticiper un risque de dépassement d'échéance à l'échelle d'une chaîne de tâches, de façon à pouvoir passer d'un Mode d'ordonnancement Nominal vers un mode Dégradé dans lequel on va prévenir la défaillance. Cette dernière a pour origine première les interférences matérielles qui augmentent les temps d'exécution des tâches. Aussi pour éviter davantage ce rallongement et donc conserver la garantie de terminaison de la chaîne de tâche avant l'échéance, on remonte à la source en prévenant temporairement tout risque d'interférences. Et cela est obtenu via un mode dégradé dans lequel la méthode consiste à stopper temporairement les tâches non critiques, cause des interférences. Il nous faut donc prendre en compte la méthode de passage en mode dégradé.

3.2.2 Passage en Mode Dégradé

Estimation de Temps d'Exécution Pire Cas restant Bien évidemment, l'estimation du Temps de Réponse Pire Cas restant est un élément clé de l'approche. Tout

l'intérêt de cette méthode réside dans la capacité à passer dans un mode dégradé. En conséquence, ce que nous nous devons de garantir, c'est le non-dépassement d'échéance sachant qu'il est possible à tout moment de prendre la décision du passage en mode dégradé, dans lequel les tâches critiques n'étant plus sujettes à interférences externes, auront un Temps d'Exécution Pire Cas bien plus faible qu'en mode Nominal. Cela implique directement que si le $rWCRT(\tau_i)$ que nous considérons est dans le contexte Dégradé, alors la détection d'un éventuel risque se fait de façon beaucoup plus permissive que si l'on considère directement les risques en mode Nominal.

Il existe plusieurs méthodes à son obtention. De façon théorique, il est possible d'exploiter les méthodes déjà existantes d'estimation de temps d'exécution pire cas, auxquelles il faut ajouter la prise en compte des temps d'activation des tâches. Ce type d'approche devient hautement dépendante du système étudié, que ce soit l'architecture matérielle, mais surtout la politique d'ordonnancement des tâches, le type de tâches (périodique, sporadique, interruption)... De façon générale, la complexité des approches théoriques n'est pas négligeable et, il faut l'admettre, hors de notre cadre d'expertise. C'est d'autant plus vrai dans un cas d'application sur processeur multicœur pour lequel il est facile de tomber dans des estimations trop pessimistes. Pour cette raison, nous avons décidé dans notre proposition d'avoir une approche plus expérimentale.

Un avantage dont nous bénéficions ici, c'est que l'hypothèse de se ramener à un cas en isolation dans le mode dégradé limite grandement les risques de variations sur les temps d'exécutions des tâches critiques. C'est ce qui permet une plus grande certitude sur les estimations expérimentales, qui ne nécessitent alors plus de couvrir toute une combinatoire incluant les tâches non critiques. L'estimation est donc faite expérimentalement en exécutant le système avec un passage forcé en mode dégradé dans lequel on peut alors mesurer sans les tâches non-critiques les Temps de Réponse Pire Cas restants $rWCRT(\tau_i)$. On notera que pour une chaîne de N tâches, il y a N-1 rWCRT à estimer. Plus de détails sur le protocole adopté pour l'estimation seront abordés en chapitre 4.

Il reste à aborder à présent la transition en elle-même vers le mode dégradé. Cette phase est importante du fait qu'elle implique des délais supplémentaires qui devront être pris en compte dans l'anticipation. De cette façon, à considérer que l'on recalcule périodiquement le risque de dépassement d'échéance, il est possible de définir l'instant où l'on sait que l'attente d'une période supplémentaire va faire que même en mode dégradé, il ne sera plus possible de garantir le respect de l'échéance bout-en-bout. Par conséquent, il devient alors clair que cet instant-là devient le dernier moment auquel il faut nécessairement passer en mode dégradé pour justement conserver cette garantie.

Transition de Mode Ce changement de mode se fait en 3 étapes. Il faut en premier lieu bien entendu la détection du point de bascule auquel le risque de défaillance est détecté, c'est l'étape de décision. Une fois la décision prise, la seconde étape est d'activer le mécanisme de passage en mode dégradé. Dans notre cas, il s'agit d'envoyer un signal aux tâches non critiques de façon à mettre en pause leur exécution, c'est l'étape de contrôle. Enfin, le système de surveillance doit continuer

d'observer l'État de la chaîne de tâche de façon à relancer les tâches non critiques une fois le risque passé. Il s'agit de l'étape de recouvrement.

Ces trois étapes font ressortir un élément important pour l'anticipation qui n'a pas été pris en compte pour le moment, et il s'agit de la durée entre l'étape de décision et la fin de l'étape d'exécution. En effet, le temps pour que toutes les tâches non critiques soient mises en pause est non nul, et ce délai doit être pris en compte dans l'estimation de Temps d'Exécution Pire Cas restant dans l'optique où le pire cas en question est considéré dans le mode dégradé.

En conclusion, en considérant les grandeurs suivantes :

— W_{MAX} La durée maximum garantie entre 2 points de surveillance de l'Etat de la chaîne de tâche

— t_{SW} Le délai maximum nécessaire à la transition du mode nominal au mode dégradé

— $rWCRT(\tau_i)$ pour chaque τ_i de la chaîne de tâche, les Temps de Réponses Pire Cas restant en mode dégradé

Il est alors possible de calculer la somme du temps de réponse partiel actuel avec ces trois métriques. Tant que cette somme reste inférieure à l'échéance bout-en-bout, alors on peut conserver le système en mode nominal. En revanche, à partir du moment où cela dépasse l'échéance, alors c'est le moment où il n'est plus sûr de rester en mode nominal, et il faut donc déclencher le mode dégradé pour garantir le respect de l'échéance. Cela correspond en conséquence à surveiller que l'inéquation suivante reste vraie pour savoir l'instant critique auquel il faut passer en mode dégradé :

$$RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW} \leq D \tag{3.1}$$

Cette équation est notamment adaptée des travaux de [Kritikakou 2014]. Chaque point de surveillance de l'État de la chaîne de tâches est considéré comme étant temporellement sûr (au sens où il n'y a pas de risque de faute temporelle due au dépassement d'échéance) tant que cette inégalité est respectée.

Démonstration. En présumant que (3.1) est respectée, on peut montrer qu'il est sûr d'attendre le prochain point de surveillance pour décider de changer de mode. Soit t_{next} le prochain instant de surveillance. Par définition, $t_{next} \leq t + W_{max}$ alors nécessairement $RT(t_{next}) \leq RT(t) + W_{max}$, par conséquent
$RT(t_{next}) + rWCRT(\tau_i) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$.
Aussi, $rWCRT()$ ne peut que décroître avec le temps qui s'écoule, par conséquent

$$rWCRT(t_{next}) \leq rWCRT(\tau_i)$$

et

$$RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$$

Étant donné que (3.1) est respecté, on a $RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq D$. De ce fait, il sera sûr de passer en mode dégradé au prochain point de surveillance. □

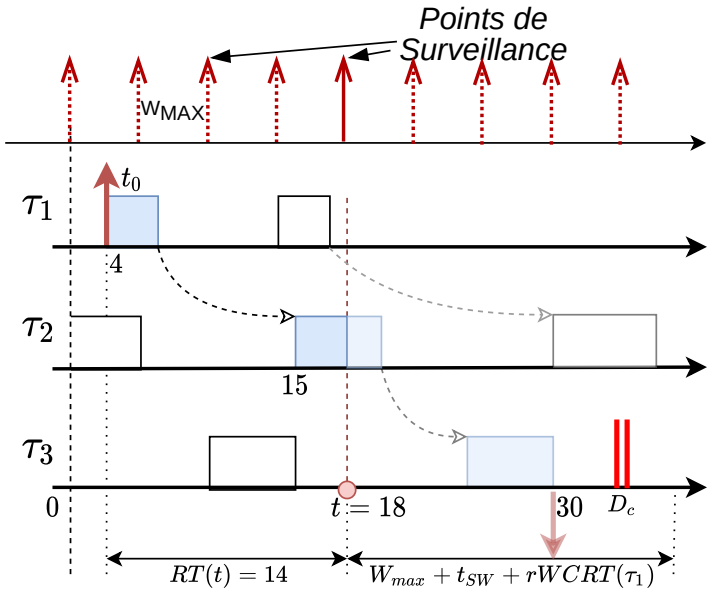


FIGURE 3.3

En adaptant ce qui a été présenté précédemment avec le risque de dépassement d'échéance, on peut retrouver en Figure 3.3 l'ajout des composants qui permettent l'anticipation du changement de mode, au regard de la période de surveillance et du temps de changement. La méthode de réglage de la période de surveillance sera discuté dans le chapitre 4.

3.3 Architecture Logicielle

Maintenant que nous avons présenté toute la logique derrière le mécanisme de surveillance et de contrôle, nous allons présenter plus en détail l'architecture logicielle nécessaire à son implémentation.

Pour résumer la situation, nous sommes dans le cas d'un système implémenté sur un calculateur multicœur, dans lequel nous avons distingué une chaîne de tâche critique des autres tâches considérées non-critiques. Il faut bien entendu ajouter à cela un Agent de Surveillance et de Contrôle pour assurer la fonctionnalité de mitigation des fautes temporelles. Ce dernier se destine à être exécuté en couche bas niveau, au même niveau que la politique d'ordonnancement du système. Dans le cadre de la suite de nos expérimentations, pour simplifier l'implémentation, nous avons considéré l'isolation de l'Agent de Surveillance et Contrôle sur un cœur du processeur. L'ensemble de ces éléments se résume en Figure 3.4.

L'Agent de Surveillance et Contrôle peut se décomposer en deux éléments distincts. D'un part un *Task Wrapper Component* (TWC) et de l'autre le *Core Control Component* (CCC). Le TWC se destine à récupérer toutes les informations nécessaires à la surveillance et la mise à jour de l'État de la Chaîne de tâche, tandis que le CCC doit prendre en compte ces informations, de façon à réaliser la prise de

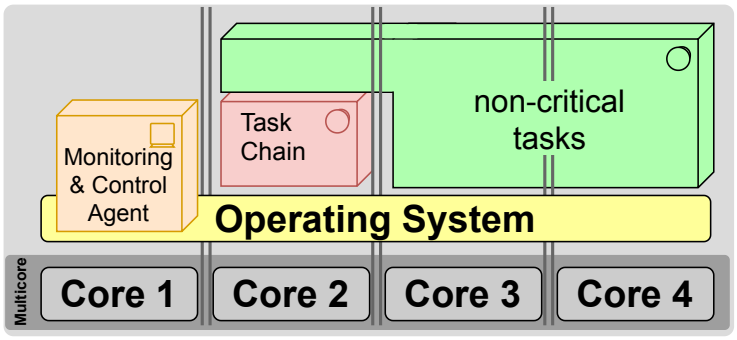


FIGURE 3.4 – Monitoring & Control Agent basic concept

décision du passage en mode dégradé au regard de l'inéquation 3.1.

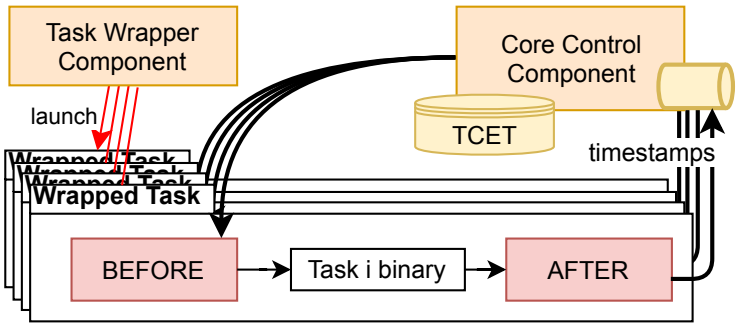


FIGURE 3.5 – Monitoring & Control Agent Architecture

3.3.1 Task Wrapper Component (TWC)

L'objectif du Task Wrapper Component est de gérer l'aspect Surveillance du mécanisme. Il englobe l'instrumentalisation du logiciel de façon non intrusive pour obtenir les informations nécessaires à la Surveillance. C'est une différence assez importante avec bon nombre de travaux qui se basent sur la surveillance de l'exécution de tâches. Souvent ces dernières reposent sur une instrumentation logicielle au niveau du code des tâches. Cela offre un suivi plus fin, mais avec un coût de développement logiciel supplémentaire pour chaque tâche. De plus c'est par définition incompatible avec du logiciel fourni en boite noire.

Le mécanisme de Surveillance et Contrôle nécessite **a)** une connaissance des date d'activation et de terminaison des jobs, **b)** ainsi que la connaissance de la structure de la chaîne de tâche . Concernant le premier point, il faut donc ajouter des blocs de code exécutés directement avant le début d'exécution des tâches ainsi que juste avant leur terminaison. En récupérant les *timestamps*[3] d'exécution de ces blocs de code, il est ainsi possible de connaître précisément les dates de début $s_{i,j}$ et de fin $e_{i,j}$ de chaque job. Ils ont vocation à être envoyés au Core Control Component pour

3. timestamp : en informatique, le temps se décompte par un compteur a une fréquence donnée. Un timestamp correspond à un temps suivant ce compteur

mettre à jour l'État de la chaîne de tâches ainsi que le suivi des Traces d'Exécution actives.

Par ailleurs, une telle instrumentalisation permet d'ajouter une couche de sécurité supplémentaire. En effet, le changement de mode se fait initialement par envoi d'un signal pour stopper les tâches non critiques. Ceci étant dit, pour une meilleure réactivité et ajouter de la redondance dans l'arrêt des tâches, il est aussi possible d'utiliser le bloc logiciel exécuté avant l'exécution des tâches non critiques pour vérifier si la tâche en question a effectivement l'autorisation de s'exécuter ou bien si on est en mode dégradé, doit être stoppée.

En somme, le Task Wrapper Component doit encapsuler les tâches pour ajouter deux wrappers : un "Before" et un "After" qui vont avoir deux rôles :

— envoyer au Core Control Component les timestamps d'exécution associés à chaque job de tâche critique (dates de début et de fin),

— ajouter de la redondance pour prévenir l'exécution des tâches non-critiques en mode dégradé et potentiellement accélérer le changement de mode.

On remarquera qu'il n'est pas utile d'exécuter un wrapper "After" à la suite des tâches non critiques étant donné que nous ne monitorons pas leur exécution dans le cadre du mécanisme.

3.3.2 Core Control Component (CCC)

Le Core Control Component gère donc l'aspect "Contrôle" du mécanisme. Il doit être exécuté périodiquement, tous les T_{CCC} unités de temps. À chaque exécution, son rôle est de récupérer toutes les données de timestamps du TWC. En ayant connaissance de la structure de chaîne de tâche qui est exécutée, il peut alors reconstruire les Traces d'Exécution telles que définies précédemment (Définition 3.2.1).

À partir des Traces actives, l'État de la chaîne de tâche $S(t)$ est alors mis à jour. Il s'agit de calculer successivement :

— $RT(t)$, la durée à partir de la date de départ de la tâche d'entrée,

— $rWCRT(\tau_i(t))$, le Temps de Réponse Pire Cas restant à prendre en compte selon l'état de la trace d'exécution active.

En y ajoutant les constantes W_{MAX} et t_{SW}, l'inégalité 3.1 est re-calculée. Dans le cas où elle est toujours respectée, rien ne se produit et le CCC peut attendre la prochaine période de vérification. Dans le cas où elle devient fausse, alors la procédure de changement de mode est enclenchée.

Il pouvait y avoir deux choix possibles d'activation du Core Control Component. Soit en le déclenchant de façon asynchrone, à chaque fois que la TWC reçoit une nouvelle donnée de timestamp, soit en le déclenchant de façon périodique indépendamment de l'exécution des tâches. Notre choix s'est donc porté sur la seconde option pour une raison fondamentale. En effet, bien qu'un déclenchement asynchrone soit plus simple à implémenter d'un point de vue technique, on perd la maîtrise de l'utilisation processeur de notre Agent de Surveillance et Contrôle, ce qui peut être dommageable pour la maîtrise de l'ordonnancement du système. De plus, dans le cas où une défaillance sur la réception des timestamps surviendrait, le CCC pourrait se retrouver dans une attente indéfinie de données, ne mettant alors pas à jour l'État

de la chaîne de tâches... et laissant donc passer les risques de faute temporelle. Par conséquent, avec un déclenchement périodique, on obtient une décorrélation du reste du système, et donc quoi qu'il arrive il sera possible à intervalle fixe de déterminer s'il y a un risque. En effet, même en l'absence de nouveaux messages la valeur de $ET(t)$ continue d'augmenter.

3.3.3 Définition des constantes de Contrôle

Il est important de fixer correctement les paramètres constants du CCC, qui sont t_{SW} and W_{MAX}. Si ces paramètres sont sous estimés, typiquement dans le cas où le temps de passage en mode dégradé est plus long, alors l'inégalité 3.1 pourra devenir tout bonnement fausse. Concernant la durée maximale entre 2 mises à jour de l'État de la chaîne de tâche W_{MAX}, la valeur est assez simple à obtenir. Du fait de la périodicité de la tâche ainsi que son très haut niveau de priorité, associé à une demande en ressource très réduit, il suffit de prendre la durée de la période fixée T_{CCC}. Là encore on réalise qu'avec une activation sporadique à l'arrivée des messages, cette durée aurait été plus complexe à fixer.

Cependant, cela repose donc sur le choix sur la fréquence d'exécution du CCC. Ce choix peut avoir plusieurs conséquences et va donc être sujet à de potentiels compromis. D'une part, un T_{CCC} trop petit signifie une plus forte utilisation des ressources de calcul, ce qui va directement à l'encontre de notre objectif d'optimiser la puissance de calcul. C'est d'ailleurs une problématique courante des mécanismes de Surveillance, d'avoir un impact négatif sur l'usage des ressources du processeur et s'avérer contre-productif.
A l'inverse, plus la période va être longue, plus le mécanisme d'anticipation sera sensible et anticipera des risques de plus en plus improbables. De fait, plus la prochaine date de vérification de l'État de la chaîne est éloigné dans le temps, plus le moindre glissement de temps d'exécution des tâches critiques va laisser penser à un risque de dépassement d'échéance. Cela fait alors augmenter le taux de faux positifs, c'est-à-dire le taux de déclenchements du passage en mode dégradé qui n'étaient pas nécessaires. Par ailleurs, il faut que le Core Control Component puisse soutenir le débit d'arrivée de données d'exécution des tâches critiques, car il n'est pas possible de maintenir en liste d'attente une quantité infinie de messages.
En conclusion, la valeur de T_{CCC} doit être choisie au regard d'une estimation du débit d'exécution des jobs, donc dépendant du nombre de tâches et de leur période d'exécution.

3.4 Application au domaine automobile (diag. fonctionnel, SWC, etc)

3.4.1 Concept Description

Our approach presents a software execution *Monitoring and Control Agent (MCA)* to guarantee end-to-end deadline constraints. We focus on the respect of end-to-end constraints of tasks chains, not individual tasks constraints. The idea

behind this is to offer more “flexibility” on tasks scheduling for guaranteeing mandatory task chains constraints if we control only end-to-end constraints instead of every critical task timing constraint. By doing so, we gain "flexibility" as we allow some parts of the chain to be behind time as they can be compensated before the end of the chain without any external action. The MCA monitors at run-time the execution time of critical tasks and anticipate when the end-to-end deadlines may be compromised to stop non-critical tasks when needed in order to avoid such risk. The anticipation is based on the estimation of remaining WCET. Finally, when the critical task chain recovers from the potential risk, the non-critical tasks can resume their execution to get back to a nominal state.

We define a *degraded mode*, opposed to the *nominal mode* of execution. In nominal mode, critical and non-critical tasks are executed normally. In Degraded mode, non-critical tasks are not executed, to prevent further interferences on critical tasks. The degraded mode implies simpler WCET estimations because we eliminate the disturbances from non-critical tasks ; such WCET will be lower than in a nominal mode. It is probably less pessimistic as we eliminate memory interferences, non-critical tasks scheduling and possible common resources (drivers for instance) usage. The main disturbances remaining will be only between the tasks from the chain. Consequently, our anticipation mechanism will be based on reduced estimation of WCET (compared to nominal mode), to activate degraded mode only as a last resort.

To reach degraded mode, MCA role is to pause/stop non-critical tasks execution. This control is triggered by an anticipation algorithm. To be efficient, this algorithm should trigger the control at the latest possible time while guaranteeing real-time end-to-end constraints.

3.4.1.1 Functional Specification

A critical task chain must describe the implementation of a system functionality from its triggering to its consequence. This would stick most of the time with a computing chain going from a sensor measure to an actuator command. First idea would be to stick with safety criticality levels (ASIL D to ASIL A and QM, for automotive applications), but we quickly notice that there is no direct link between this classification and critical tasks chains. A safety critical task is not necessarily defined from its timing constraints. The only possible conclusion here is that a critical task chain only includes non-QM tasks.

We propose here a definition based around high-level specifications as represented in figure 3.6. The global system is defined as a set of features [4]. Every feature gathers a set of functionalities that are translated into Use Cases [5]. A Use Case defines a feature behavior for a given context and inputs (and the consequent outputs). Finally, those are translated into functional chains representing different functions and their interactions needed for the realization of the Use Case.

4. Features : all the services the system must provide. e.g : Lane Support System (LSS) is a feature.

5. e.g : Lane Departure Warning & Lane Keeping Assist are part of the use cases of LSS feature.

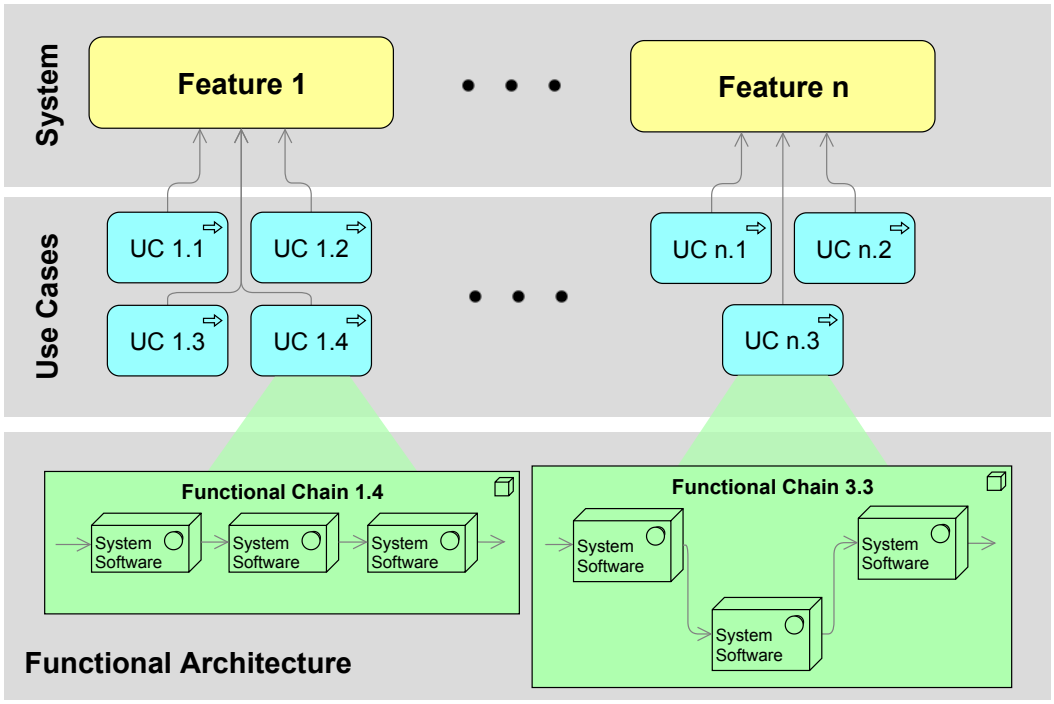


FIGURE 3.6 – Functional Architecture definition

If we combine this information with a severity classification in case of failure of the use cases, it is possible to define critical chains as functional chains with a high severity risk. This is one possible criterion allowing an easy separation between a critical functional chain and the others. It could be adapted during the design phase, depending on the functional chains allocated to the processor.

Such information allows to define the software components involved in the critical task chain. All the software components used to realize a critical functional chain form a critical task chain at an OS point of view. At this point, it is possible to define the task chain end-to-end deadline, following the severity temporal risk in case of failure. Such deadline should be at minimum the sum of individual tasks deadline, but could probably be higher, depending on the global system and the task chain function. Our objective is to guarantee such critical task chain end-to-end execution time on the multicore.

Chapitre 4

Protocole et démarche expérimentale

Sommaire

4.1 Principe Général et Objectifs

We present in this section the experimental protocol proposed to characterise the system tasks (the “workload") and calibrate the Monitoring and Control agent. The experimental protocol is divided in 7 steps separated in 3 phases : 1. Design phase, 2. Calibration phase, and 3. Run-time validation phase as resumed in Tableau 4.1.

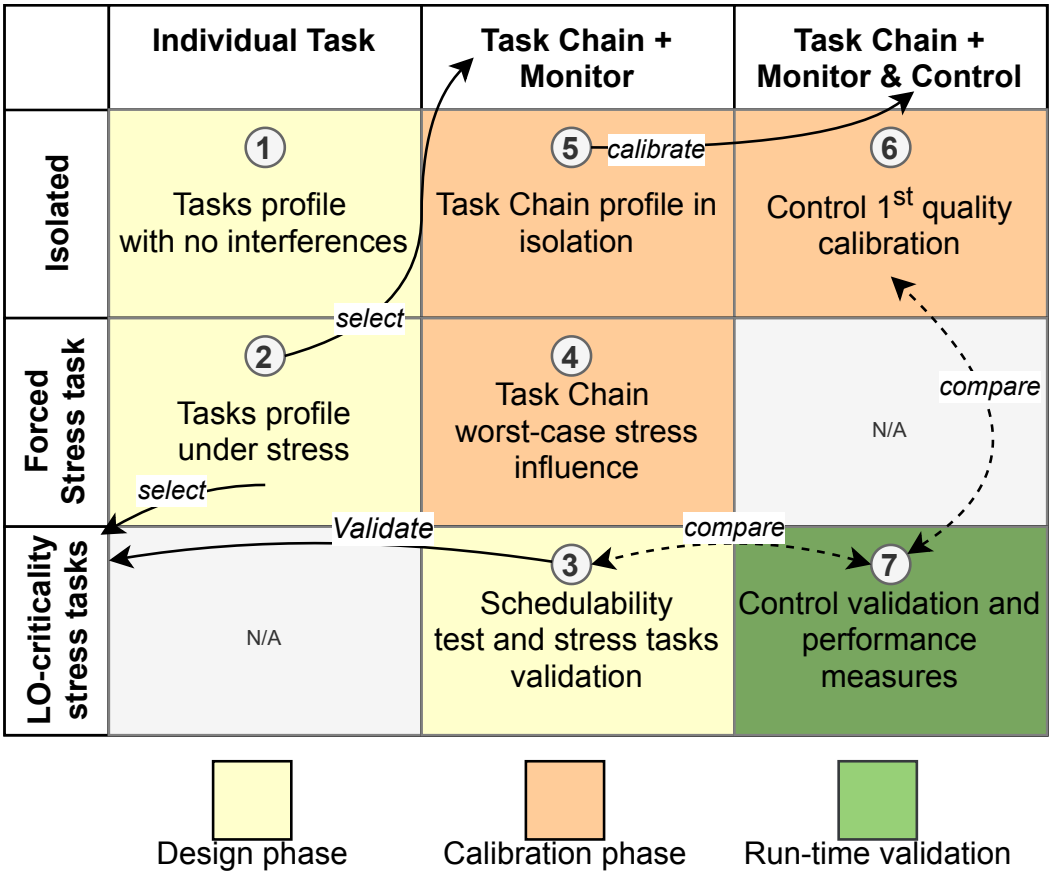
The experimental steps are incremental following two inputs, final step being the complete system with task chain monitoring and control. The first input is the functional load under test that is executed on the real-time framework. It can be either :

1. single task : a specific task from the workload is executed ;
2. task chain : a specific task chain, made of multiple tasks, is executed and monitored ;
3. Task Chain with Monitoring & Control mechanism.

Second input is the system load executed along with the functional load to influence its execution. It can be either :

1. none, to test an isolated functional load ;
2. forced stress : strong cache, memory and CPU stress from linux `Stress-ng` tool set ;

TABLE 4.1 – Experimental Flowchart



3. real-time tasks : the LO-criticality tasks of the workload are executed.

Those inputs results in a two entry table as shown in Tableau 4.1 with each box corresponding to a step in the protocol.

4.2 Phase de Design

This phase is needed if the workload involved don't come with a detailed specification, including their behavior and execution times. This is the case of our experiments as we select tasks from an already existing benchmark and we have no information about tasks execution times or even their compatibility with our real-time environment. Thus this phase is to characterise the available task set and define the workload specifications. It will be split into the HI-criticality task chain and LO-criticality tasks with their characteristics (min/avg/max execution time, periodicity...). This phase is defined by steps ①, ②, ③ in Tableau 4.1.

4.2.1 Profil des tâches en isolation

First objective is to get a global idea of tasks execution time profiles. One experiment is made per task, the task being executed individually with the framework. The task is called periodically with a given input, and task response times are logged.

4.2.2 Profil des tâches avec stress imposé

We add to the precedent step an artificial system load to cause high stress on cache, memory, I/O and computing use while the tasks are executed one by one. The output is a table with a profile for each task made of the min/average/max execution times and system metrics (system calls, context switches, scheduling interrupts, eventual period misses...). Such profile allow to categorise the tasks following their sensitivity to interferences compared to previous step ①. This allows to define which tasks can be used for the HI-criticality task chain or as stressing LO-criticality tasks but also discard any task that would not fit our needs.

4.2.3 Chaine de tâches avec système complet sans Contrôle

Previous step classified the task set between HI and LO-criticality tasks. We define on this step the specific task chain and LO tasks that will be studied next and verify the pertinence of such choice. We check the workload schedulability in the soft real-time sense (i.e. schedulable if deadlines tardiness are bounded by a reasonably small constant). We also measure the task chain response time profile under “realistic" conditions without the Control mechanism enabled. Expected result is a schedulable system with reference task chain response times with interferences.

4.3 Phase de Calibration

This phase is mandatory to configure the Control mechanism to the software and hardware specificities and lower false-positive rate. It is made of steps ④, ⑤, ⑥ in orange boxes of Tableau 4.1. Configuration includes task chain worst-case response time and intermediary response times in isolation. Performance optimisation consist in tweaking the switch time t_{sw} and anticipation execution frequency W_{max} constants, in the objective of lowering false-positive anticipation rates.

4.3.1 Chaine de tâches avec stress forcé

The task chain is then tested under a worst-case scenario. It is executed with the artificial system load, to stress as much as possible the task chain similarly to step ②. We get a baseline of the worst-case chain response time. This value is important because if the end-to-end deadline is always greater than the worst-case response time observed then the mechanism would be of no use (i.e. deadline never broken from temporal faults). This step gives a quantification of the task chain sensitivity to interferences and thus indicates the pertinence of using a Monitoring and Control Agent to manage them.

4.3.2 Chaine de tâche en isolation

The objective is to calibrate Control mechanism parameters : $rWCRT(\tau_i)$, Core Control Component period (T_{CCC}) and switch time (t_{sw}) to degraded mode. The task chain is executed alone with the MCA but with the Control mode switch disabled. We log every chain intermediary and end-to-end response times. The result gives the data of all the remaining response times obtained during the test. We set the $rWCRT(\tau_i)$ parameters as an upper limits of the remaining response times registered.

4.3.3 Chaine de tâche avec mécanisme de Contrôle

Finally, the Control mechanism is enabled, with the parameters set on previous step. As this step does not include the LO tasks that bring interferences to the task chain, the Core Control Component should not trigger any switch to degraded mode. This step is important for the final analysis as it already points out the base false positive rate obtained with chosen parameters. A qualitative MCA should have the least degraded mode switch possible. Otherwise it could mean that either the CCC parameters are not ideally set (typically W_{max}), or the expected timing delays caused from interferences are too close to the usual timing variation of the task chain execution even in isolation. In other words, the Control Component is not able to differentiate response time variations due to temporal faults from ones due to nominal execution time variations. Another possibility is the end-to-end deadline requirement is too close to the nominal end-to-end response time in isolation.

4.4 Phase de Validation en exécution

4.4.1 Chaine de tâches avec système complet et mécanisme de Contrôle

The validation phase implies a last step (⑦ in green box of Tableau 4.1), which is with the whole final system being executed : HI task chain and LO tasks with the MCA enabled. The objective is to collect the concluding information on the Monitoring and Control Agent behavior to measure the 3 quantification criteria (efficiency, performance and quality) of the solution explained in **??**. We also use the data from steps ③ and ⑥ as a reference for the conclusions.

Chapitre 5

Cas d'implémentation de l'Agent de Monit. & Contrôle

Sommaire

5.1 Framework et Architecture Logicielle

5.1.1 Plateforme Matérielle

The platform used for the experimentation is a barebone computer equipped with a processor Intel Core i5-8250U. This processor embeds 4 cores. It has 3 caches level, L1, L2 and L3 (shared), with respectively 32 KiB/core, 256 KiB/core and 8 Mib (shared). We fixed its frequency to 1400MHz and disabled hyper-threading for our tests.

5.1.2 Support Logiciel

We used Linux (Linux Mint xfce 18.04 distribution) to mix general purpose and real-time applications with different scheduling policies ([Wong 2008], [Lelli 2011]). Its versatility grants easier compatibility with benchmarking suites. Moreover, by adding Xenomai (v. 3.1) real-time co-kernel [Gerum 2004], it is possible to get closer to real-time applications with latencies lowered from milliseconds down to microseconds. It also grants an API for real-time application development, used for the MCA framework.

Notably, POSIX enables to force tasks execution to dedicated cores and change both priority and scheduling policy. As we are in a controlled context that suppose no malicious behavior, we do not implement mechanisms like memory protection or strong space isolation policies. As stated before, vanilla Linux Kernel is not made for hard real-time application. That is mainly because kernel is not preemptive on

most parts of it, this can cause high latency for real-time interrupts, from kernel code execution that could be linked to non-critical applications. Therefore, we add a Xenomai co-kernel to improve latency down to micro-seconds and run our MCA to respect desired real-time constraints. Please note that from Linux point of view, “threads" and “processes" are equivalent and correspond to “tasks" for us.

Threads are assigned 2 parameters, a scheduling policy and a static priority (*sched_priority*). Both are considered by the global scheduler. It first gathers the threads by priority level to execute highest priority processes first. Then for a same priority level, the scheduling policy of each task will define which one to run first. For normal processes the priority level is ignored (considered at 0) to be executed following the CFS policy. This way, a real-time process with a priority level from 1 (lowest) to 99 (highest), always run before them. The threads' scheduling policy defines how they are inserted into the list of same priority level and how they move in this list, all processes being preemptive. We can list 3 real-time policies for real-time process : FIFO, EDF and Round-Robin.
For this purpose, Linux allows to bound threads to cores. For a processor with j cores, every thread has a core affinity represented by an array of j Booleans. Each of these Booleans of affinity b_{Ti} indicates if the thread T can be executed on the core i. By default, every normal process has a core affinity of $\begin{pmatrix}1 & 1 & 1 & 1\end{pmatrix}$, for a quad-core processor, meaning that it can be executed by every core. It makes it easier for the scheduler to balance load between every core. But for our case and when it comes to run hard real-time applications, it is interesting to use such affinity. This way, it will be possible to isolate our MCA on an isolated core and bound the benchmark processes to the other threads. Xenomai is a real-time kernel that can be installed as a co-kernel to a classic Linux distribution as presented in deep by [Gerum 2004]. Our framework and experiments are implemented on the real-time APIs proposed by Xenomai 3.1. In such configuration, it adds an interruption pipeline (ADEOS) directly between the hardware and OS low-level software (i.e. Hardware Abstraction Layer, OS Kernel and drivers). This enables to catch all the interrupts and distribute them in priority to Xenomai real-time kernel. Threads executed with Xenomai are executed either in primary or secondary mode. In both cases they are memory-protected from other processes. By default, Xenomai threads starts in primary mode. They get directly access to Xenomai API and are scheduled by its real-time scheduler. A Xenomai thread can however use kernel API with system calls. When it happens, the Xenomai tasks goes temporarily to the Linux scheduler and automatically goes back to Xenomai domain once done. As the priority system used on primary mode is compatible with the secondary one, the Xenomai tasks keep their highest priority status. It makes the mode switch transparent.

All things considered, we mainly use Xenomai to get a significant latency gain (divided by up to 10) for the critical tasks. We can stay on the classic Linux domain for our non-critical tasks.

Such OS configuration allows us to specify a per-task core allocation and priority level. Linux scheduler as explained in [Ishkov 2015] selects tasks first by priority level, (from 1 to 99 for real-time tasks domain). Then for a given priority level,

TABLE 5.1 – MiBench selected tasks

Automotive	basicmath, bitcount, qsort, susan (smooth/edges/corners)
Network	dijkstra, patricia
Consumer	jpeg (code & decode), typeset
Office	stringsearch
Security	blowfish, rijndael, sha
Telecom	adpcm (coding & decoding), CRC32, FFT, gsm

multiple scheduling policies are possible : Global Earliest Deadline First, FIFO, Round-Robin, and other best-effort policies. To test a system using classic Round-Robin for instance, every task are launched at same priority level with Round-Robin policy. We use Rate-Monotonic scheduling policy for our tests this way.

5.2 Benchmark MiBench

5.2.1 Présentation

MiBench [Guthaus 2001] plays the role of the task set to constitute our experimental workload. This benchmark suite gives source code for 30+ standalone binaries classified in six domains : automotive, security, network, telecommunication, office and consumer. Those tasks do different jobs similar to ones in these domains, with different levels of complexity that is of high interest for us.

To run an artificial system load as a “worst-case" cache, memory, CPU use and I/O stress, we use Linux *Stress-ng* tool presented in [King 2019].

As we do not have yet real industrial application for testing, for now the MiBench Benchmark suite [Guthaus 2001] has been used for our experiments. The objective is to use applications similar as much as possible to computation profiles that could be found in real applications, in order to reproduce memory containment and resource usage close to real cases.

MiBench consists of a large panel of tasks with different memory needs and execution profiles to mimic existing applications. We have at disposal applications from 5 different domains, as presented in the Tableau 5.1. It is used here to validate the framework and put into practice our experiments.

We selected a set of 16 applications from MiBench for our experiments. Most of them exists in “small" and “large" version that allows to change proportionally their execution time and resource needs. Also, some of these tasks may have several variants according to setup parameters. For instance, *Sunsan* has 6 different variants : edge detection, corner detection and smoothing, all 3 existing in both “small" and “large" version which works with a bigger image for processing. This way, those 16 applications leads to 45 different possible tasks for our experiments. It enables to test different combination following the “size" and number of tasks but also the kind of tasks we use. Tasks profile classification were already made by Guthaus &

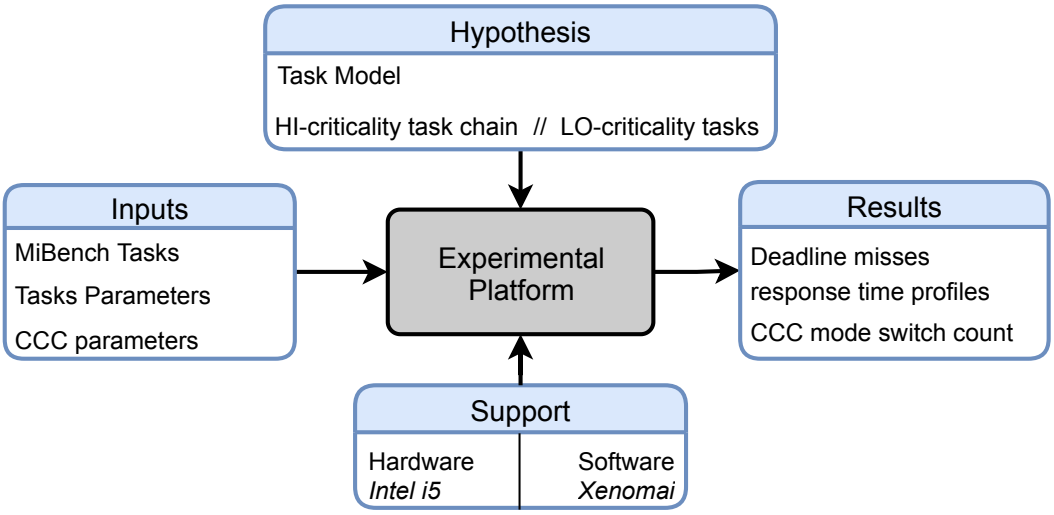


FIGURE 5.1 – Experimental Platform structure

al. in [Guthaus 2001] and detailed work about their memory consumption can be found in [Blin 2016].

5.2.2 Demandes d'adaptation/modification des tâches

5.3 Agent de Monitoring et Control

5.4 Solutions adoptées à la complexité d'implémentation

Difficultés rencontrées dans la mise en place de ce concept et leçons apprises (en cas de volonté de reproduction)

Chapitre 6

Mise en Application expérimentale

Sommaire

6.1 Application à MiBench du Protocole

Using Mibench as a workload had advantages but also drawbacks. It allows to get specific tasks with a defined and already studied behavior but we are dependent on the way they are initially programmed. They might not completely fit our needs to simulate embedded applications or have incompatibilities with the chosen real-time environment. First step in using this benchmark is to check those criteria to select precisely the tasks from MiBench we use.

6.1.1 Phase de Design

6.1.1.1 Profil des tâches en isolation

We need to establish the execution time profile of each task of the bench. As a result some tasks will be removed from the tests, either due to execution time magnitude differences or inconsistent behaviors between experiments. Accordingly, we measure on each experiment the min, max and median execution times, but also some system counters as the Xenomai mode switches and the amount of linux system calls. Without interferences, the execution time characteristics should have low variations.We see in Tableau 6.1 a sample of the tasks characteristics collected, for 3 different profiles.

With such data, we identified the majority execution time range in MiBench task set around 10ms (from 2-3ms to 20-30 ms) and the basic system calls and mode switch amounts due to initialisation phase (respectively 58 mode switches and ≈hundreds of system calls).

Consequently, we discard tasks out of the execution time magnitude like *adpcmCaudio_L* with an average execution time of 432 ms. By the end

TABLE 6.1 – Tasks profiles in *Xenomai* environment

Task	execution times (ms)		System Counters	
	Median	Max	Mode Switch	Sys. Call
Patricia	0.026	0.099	10051	10338
FFT	7.36	7.39	58	2343
rijndaelE	140,11	141.81	158	446

of step ①, we retained 34 tasks : Bitcount_L, Bitcount_S, Basicmath_S, Basicmath_L, Dijkstra_L, Dijkstra_S, Fft_inv_L, Fft_inv_S, Fft_L, Fft_S, GsmToast_L, GsmToast_S, GsmUToast_L, GsmUToast_S, RijndaelE_S, RijndaelD_S, Sha_L, Sha_S, Stringsearch_L, Stringsearch_S, AdpcmCaudio_L, AdpcmCaudio_S, AdpcmDaudio_L, AdpcmDaudio_S, Cjpeg_L, Cjpeg_S, Djpeg_L, Djpeg_S, Susan_L_corners, Susan_S_corners, Susan_L_edges, Susan_S_edges, Susan_L_smooth, Susan_S_smooth.

6.1.1.2 Profil des tâches avec stress imposé

We add stress on cache level and communication bus from previous step experiments. The objective is to discriminate our tasks in two groups depending on their reaction under stress. If it increases execution time too significantly (more than x10 from average time in isolation) it means the tested task is not suited for the tested environment and suffers not only from interferences but also from LO-criticality tasks preemption. A significant increase in mode switches also indicates such behavior. The tasks that do not pass correctly this test will be either ignored or used LO-criticality stress tasks. Tasks without an exploding execution time or huge increase of mode switches will be used to generate the HI-criticality task chain. Execution time profiles of task used for this purpose are in Tableau 6.2. We finally retained 22 tasks at the end of step ②.

TABLE 6.2 – Tasks profiles in *Xenomai* environment

Task	execution times isolated		execution times stressed	
	Median (ms)	Max (ms)	Median (ms)	Max (ms)
djpeg	1.97	2.28	19.91	211.53
rjindaelD	8.80	9.77	35.02	526.33
FFT	1.85	1.86	2.03	14.8
FFT^{-1}	3.56	3.57	4.05	19.74
bitcount	8.36	9.52	9.98	45.18

6.1.1.3 Chaine de tâches avec système complet sans Contrôle

At this point, we defined our task set, composed of the LO-criticality tasks used as “real" stress and the task chain made of 5 tasks :

$$FFT \rightarrow Bitcount \rightarrow Basicmath \rightarrow FFT^{-1} \rightarrow sha.$$

We need to verify the validity of our choice in term of schedulability and effectiveness of the LO-criticality tasks as interferences. Executing the whole task set together allow to verify both for this step ③.

The right part (blue) of **??** shows the task chain response time distribution profile with the full workload executed (i.e. LO-criticality tasks included). We see the perturbation due to the LO tasks on the critical task chain execution. Our workload is schedulable (no execution drops and deadline misses have reasonable overheads) and the task chain meets high response times compared to its average "nominal" response time for $\approx 10\%$ of the executions (above 200ms response time). We arbitrarily define the task chain deadline $D = 160$ms.

6.1.2 Phase de Calibration

This phase is dedicated to configure the Core Control Component parameters ($rWCRT_i(\tau_i)$, t_{sw} and W_{max}) and run the reference experiments of the task chain behavior on a worst-case stress context (step ④).

6.1.2.1 Chaine de tâches avec stress forcé

In this part we use *Stress-ng* to simulate a worst case stress condition. The task chain potential worst case response time in this context raises at 300ms. Such increase by 100% of the max chain response time under this scenario indicates the pertinence of using a MCA. Regarding such result, our workload stresses the task chain in a significant magnitude.

6.1.2.2 Chaine de tâche en isolation

For step ⑤, we execute the task chain in isolation (i.e. degraded mode). Execution time profile is on the left part (blue) of **??**. We calibrate the Monitor & Control mechanism parameters. We need the different $rWCRT$s for each value of τ_i as defined in **??**. For such linear 5-task chain we logically have $i \in \{1, 5\}$. At run-time, the remaining response times are logged in degraded mode, i.e. the task chain in isolation, and we keep an upper value of the worst measured remaining response time for each τ_i as its $rWCRT(\tau_i)$ in Tableau 6.3. Finally, regarding previous results from step ③, we set $W_{max} = 1$ms, and $t_{sw} = 500\mu$s for our platform.

TABLE 6.3 – Task Chain $rWCRT(\tau_i)$ values in degraded mode

$rWCRT$	τ_0	τ_1	τ_2	τ_3	τ_4
time (ms)	129	93	68	49.5	25

6.1.2.3 Chaine de tâche avec mécanisme de Contrôle

With the previous calibration, we can execute the task chain alone with the Control mechanism enabled. In this isolation case, we should see almost no switch

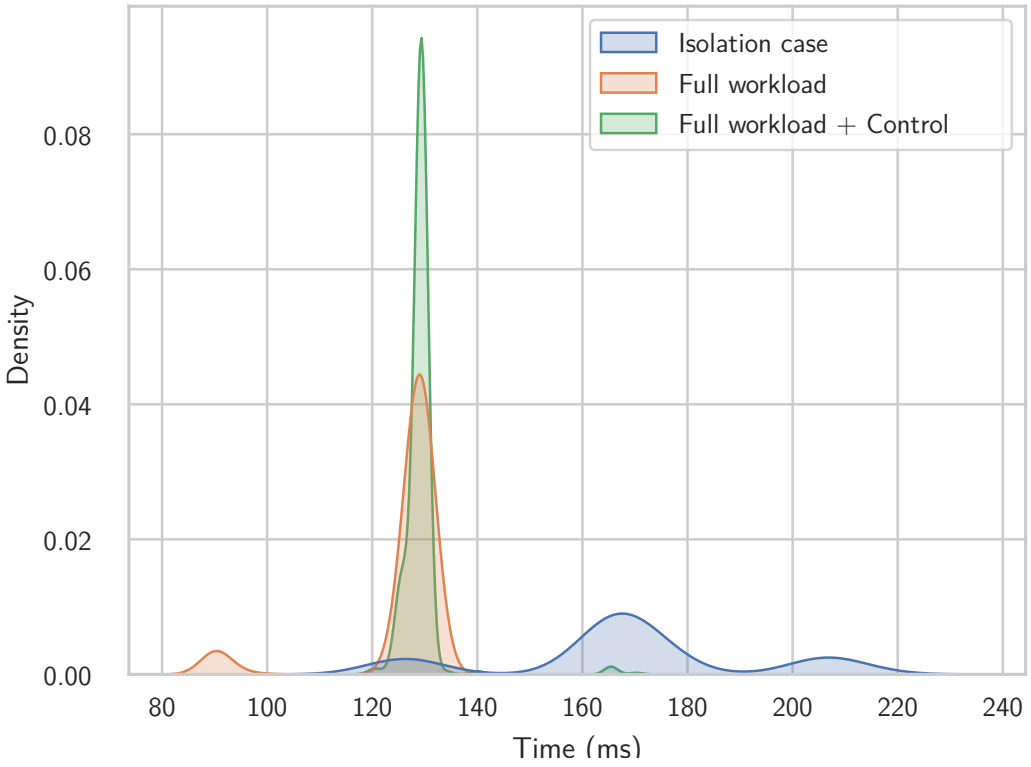


FIGURE 6.1 – Task Chain response time profile from steps ③, ⑥, ⑦

to degraded mode (and on a perfect case, no switches at all) as they must be false-positive. This experiment allows to validate the parameters set on the previous step. On our tests, we measured 0.3% of false positive triggers to degraded mode. The task chain in degraded mode response time distribution profile is illustrated in Figure 6.1.

6.1.3 Phase de Validation en exécution

6.1.3.1 Chaine de tâches avec système complet et mécanisme de Contrôle

As a final experiment, we test the complete workload (HI and LO tasks) with the Monitoring & Control Agent enabled and configured from previous step. First we observe the MCA CPU use, that is inferior to 1%. For a 120s long experiment, it ran for 1.3s overall (including setup time). We were not able to find any difference regarding CPU percentage use with and without our mechanism, either with a big task sets (small tasks only, CPU usage around 80% displayed) and with smaller task sets (e.g. only the task chain described above). Such footprint is low enough to include easily such mechanism.

In term of **efficiency**, our MCA prevented every task chain execution over a 170ms response time. Only 6 occurrences (0.1%) missed the deadline set at 160ms. The MCA brought down the average response time of the chain from 168ms (no

Control enabled) to 129ms. Such value is way closer to the average task response time profile in isolation (125ms). The few missed deadlines can be explained by the implementation framework we used, with a workload (MiBench tasks) not fully compliant with real-time programming constructs recommendations that causes uncontrolled linux system calls for instance. In conjunction with the exacting deadline we arbitrarily set at 160ms while the general workload is demanding (generating 84% deadline misses without the MCA in step ③), this explains this non-perfect result. We could use more pessimistic $rWCRT(\tau_i)$ values to achieve no deadline misses, at the expense of a worse result on the quality criteria. By the end it is a question of compromise, depending on the specific needs.

The **quality** of our calibration seems promising as there were less switches to degraded mode with the Control enabled than the number of deadline misses with no Control at all. This implies that preventing a deadline miss had a more general impact reducing the overall number of timing faults.

In term of **performance**, the system maintained LO-criticality mode for 82s / 120s total, i.e. a performance factor of 0.69 for a loss of 31% of the time in degraded mode.

All those metrics are promising for the use of a Monitoring and Control Agent in order to change a chain response time at an optimum value to avoid the great majority of the deadline misses and on the same time still take few compromises on the LO-criticality tasks execution.

6.2 Conclusions expérimentales

Conclusion et Perspectives

6.3 Conclusion

6.4 Perspectives et améliorations possibles

6.4.1 Mode dégradé multi-niveau

6.4.2 mode dégradé par mécanismes de contrôle hardware

Annexe A

Exemple d'annexe

A.1 Exemple d'annexe

Bibliographie

[2018] , I. . S. *Road vehicles — Functional safety — Part 7 : Production, operation, service and decommissioning*, 2018. (Cité en page 18.)

[Avizienis 2004] Avizienis, A., Laprie, J.-C., Randell, B. et Landwehr, C. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pages 11–33, 2004. (Cité en page 11.)

[Baufreton 2010] Baufreton, P., Blanquart, J., Boulanger, J., Delseny, H., Derrien, J., Gassino, J., Ladier, G., Ledinot, E., Leeman, M., Quéré, P. et Ricque, B. *Multi-Domain Comparison of Safety Standards*. Embedded Real Time Software & Systems (ERTS2), 2010. (Cité en page 18.)

[Blanchet 2016] Blanchet, M. *Industrie 4.0 : nouvelle donne industrielle, nouveau modèle économique*. Géoéconomie, vol. 82, no. 5, page 37, 2016. (Cité en page 4.)

[Blin 2016] Blin, A., Courtaud, C., Sopena, J., Lawall, J. et Muller, G. *Understanding the Memory Consumption of the MiBench Embedded Benchmark*. Dans International Conference on Networked Systems, pages 71–86, Marakech, Morocco, 2016. (Cité en page 52.)

[Blin 2017] Blin, A. *Vers une utilisation efficace des processeurs multi-coeurs dans des systèmes embarqués à criticités multiples*. PhD thesis, Université Pierre et Marie Curie - Paris VI, Paris, 2017. (Cité en page 23.)

[Durrieu 2014] Durrieu, G., Faugere, M., Girbal, S., Pérez, D. G., Pagetti, C. et Puffitsch, W. *Predictable Flight Management System Implementation on a Multicore Processor*. Dans Embedded Real Time Software (ERTS'14), 2014. (Cité en page 7.)

[Friese 2018] Friese, M. J., Ehlers, T. et Nowotka, D. *Estimating Latencies of Task Sequences in Multi-Core Automotive ECUs*. 2018. (Cité en page 29.)

[Gerum 2004] Gerum, P. *Xenomai - Implementing a RTOS Emulation Framework on GNU/Linux*. Rapport technique, Xenomai, 2004. (Cité en pages 49 et 50.)

[Graydon 2013] Graydon, P. et Bate, I. *Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling*. Dans WMC, 2013. (Cité en page 27.)

[Guthaus 2001] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T. et Brown, R. B. *MiBench : A Free, Commercially Representative Embedded Benchmark Suite*. Dans 4th International Workshop on Workload Characterization, Austin, TX, USA, 2001. IEEE. (Cité en pages 51 et 52.)

[IEC 61508 2010] IEC 61508. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. Rapport technique IEC 61508, The International Electrotechnical Commission, 2010. (Cité en page 16.)

[Ishkov 2015] Ishkov, N. *A Complete Guide to Linux Process Scheduling*, 2015. (Cité en page 50.)

[King 2019] King, C. I. *Stress-Ng - A Stress-Testing Swiss Army Knife*, 2019. (Cité en page 51.)

[Kotaba 2013] Kotaba, O., Nowotsch, J., Paulitsch, M., Petters, S. M. et Theiling, H. *Multicore in Real-Time Systems–Temporal Isolation Challenges Due to Shared Resources.* Dans 16th Design, Automation & Test in Europe Conference and Exhibition, 2013. (Cité en page 14.)

[Kritikakou 2014] Kritikakou, A., Pagetti, C., Baldellon, O., Roy, M. et Rochange, C. *Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems.* Dans 26th Euromicro Conference on Real-Time Systems (ECRTS14), pages 119–128. IEEE, 2014. (Cité en page 36.)

[Laprie 1996] Laprie, J., Arlat, J., Blanquart, J., Costes, A., Abdeddaim, Y., Deswarte, Y., Fabre, J., Guillermain, H., Kaaniche, M., Kanoun, K., Mazet, C., Power, D., Rabejac, C. et Thevenod, P. Guide de la sûreté de fonctionnement. Cépaduès-Editions, France, 1996. (Cité en page 10.)

[Lelli 2011] Lelli, J., Lipari, G., Faggioli, D. et Cucinotta, T. *An Efficient and Scalable Implementation of Global EDF in Linux.* 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11), 2011. (Cité en page 49.)

[Owens 2008] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. et Phillips, J. C. *GPU Computing.* Proceedings of the IEEE, vol. 96, no. 5, pages 879–899, 2008. (Cité en page 9.)

[Rupp 2020] Rupp, K. *42 Years of Microprocessor Trend Data | Karl Rupp*, 2020. (Cité en page 6.)

[Schmidt 2010] Schmidt, A., Dey, A. K., Kun, A. L. et Spiessl, W. *Automotive User Interfaces : Human Computer Interaction in the Car.* Dans Extended Abstracts on Human Factors in Computing Systems, pages 3177–3180. ACM, 2010. (Cité en page 4.)

[Smotherman 2005] Smotherman, M. *History of Multithreading.* Retrieved on, pages 12–19, 2005. (Cité en page 6.)

[TC22/SC3/WG16 2011] TC22/SC3/WG16, I. *ISO 26262 : Road Vehicles — Functional Safety*, 2011. (Cité en page 16.)

[Thompson 2006] Thompson, S. E. et Parthasarathy, S. *Moore's Law : The Future of Si Microelectronics.* Materials Today, vol. 9, no. 6, pages 20–25, juin 2006. (Cité en page 6.)

[Vestal 2007] Vestal, S. *Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance.* pages 239–243. IEEE, 2007. (Cité en page 27.)

[Wilkes 1965] Wilkes, M. V. *Slave Memories and Dynamic Storage Allocation.* IEEE Transactions on Electronic Computers, no. 2, pages 270–271, 1965. (Cité en page 7.)

[Wong 2008] Wong, C. S., Tan, I., Kumari, R. D. et Wey, F. *Towards Achieving Fairness in the Linux Scheduler.* SIGOPS Oper. Syst. Rev., vol. 42, no. 5, pages 34–43, 2008. (Cité en page 49.)

Résumé : resume **Mots clés :** mots, clefs